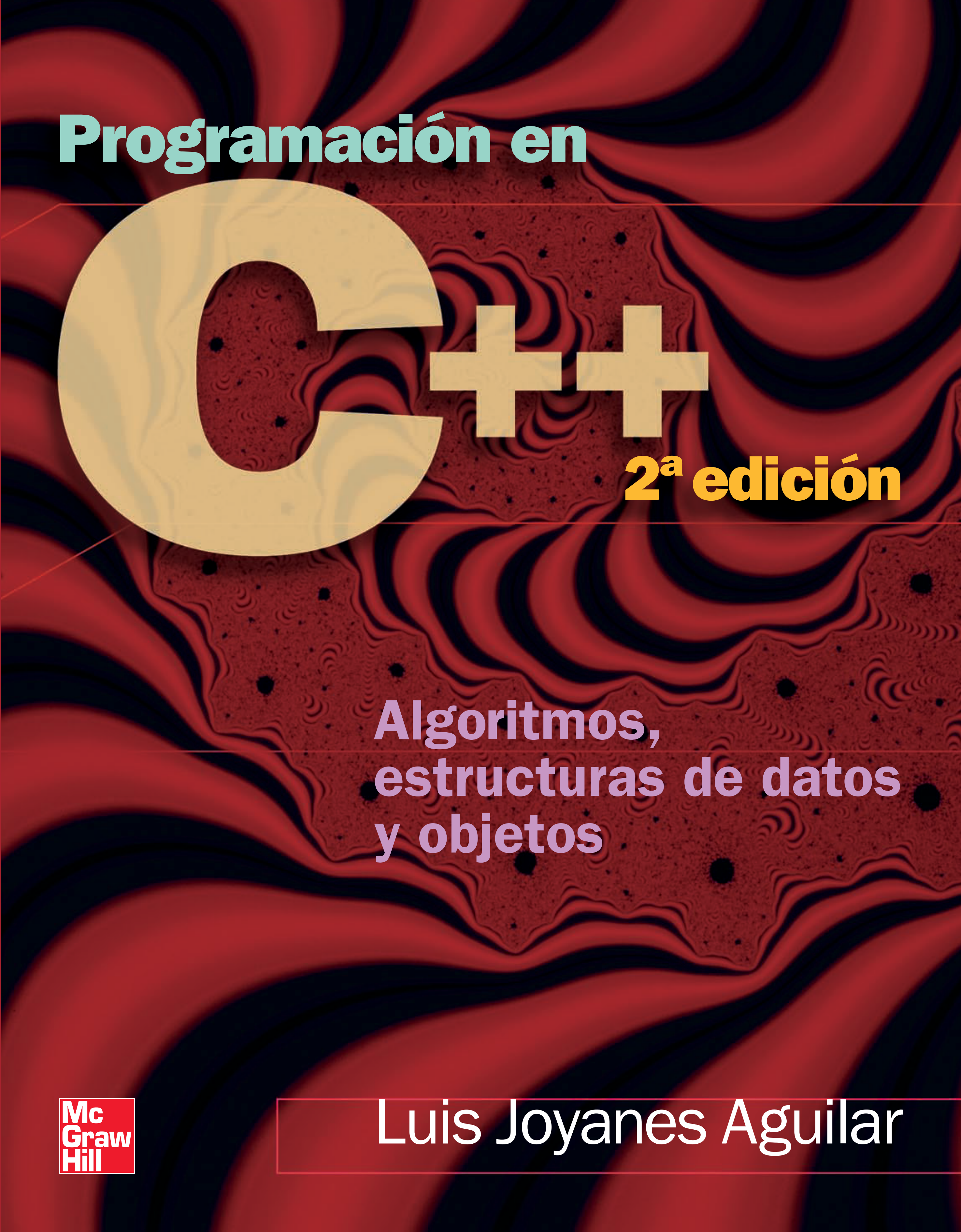


Programación en
C++
2ª edición
Algoritmos,
estructuras de datos
y objetos
Mc
Graw
Hill
Luis Joyanes Aguilar

Programación en C++

Algoritmos, estructuras de datos y objetos

2.ª edición

Programación en C++

Algoritmos, estructuras de datos y objetos

2.ª edición

Luis Joyanes Aguilar
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática, Escuela Universitaria de Informática
Universidad Pontificia de Salamanca *campus* Madrid



MADRID • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • SAN LOUIS • TOKIO • TORONTO

The McGraw·Hill Companies

Programación en C++. Algoritmos, estructuras de datos y objetos. 2.ª edición



McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1.ª planta
Basauri, 17
28023 Aravaca (Madrid)

www.mcgraw-hill.es
universidad@mcgraw-hill.com

ISBN: 84-481-4645-X
Depósito legal: M.

Editor: Carmelo Sánchez González
Diseño de cubierta: Luis Sanz Cantero
Compuesto en Puntographic, S. L.
Impreso en

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Contenido

Prólogo a la segunda edición

INTRODUCCIÓN

Esta segunda edición de ***Programación en C++. Algoritmos, Estructura de datos y Objetos,*** se ha escrito, al igual que la primera edición, pensando en que pudiera servir de referencia y guía de estudio para un primer *curso de introducción a la programación*, con una segunda parte que, a su vez, sirviera como continuación y de *introducción a las estructuras de datos* y *a la programación orientada a objetos*; todos ellos utilizando C++ como lenguaje de programación. Sin embargo, esta segunda edición ofrece novedades que confiamos mejoren sensiblemente a la primera edición. En particular, se ha tenido en cuenta la opinión de lectores, profesores y maestros, que han utilizado la obra, en su primera edición, como referencia, bien en forma de libro de texto, bien en forma de libro de consulta o simplemente como libro complementario y que sus aportaciones nos ha ayudado considerablemente. Al igual que en la primera edición, el objetivo final que se busca es, no sólo describir la sintaxis de C++, sino, y sobre todo, mostrar las características más sobresalientes del lenguaje a la vez que se enseñan técnicas de programación estructurada y orientada a objetos, pero en este caso utilizando el estándar **ANSI/ISO C++.** Por consiguiente, los objetivos fundamentales son:

- Énfasis fuerte en el análisis, construcción y diseño de programas.
- Un medio de resolución de problemas mediante técnicas de programación.
- Actualización de contenidos al último estándar ANSI/ISO C++, incluyendo las novedades más significativas (tales como espacio de nombres, archivo iostream...).
- Tutorial enfocado al lenguaje, mejorado con numerosos ejemplos, ejercicios y herramientas de ayuda al aprendizaje.
- Descripción detallada del lenguaje, con un énfasis especial en técnicas de programación actuales y eficientes.
- Reordenar el contenido en base a la experiencia adquirida en la primera edición.
- Una introducción a la informática, a las ciencias de la computación y a la programación, usando una herramienta de programación denominada C++.

En resumen, éste es un libro diseñado para enseñar a programar utilizando C++, no un libro diseñado para enseñar C++, aunque también pretende conseguirlo. No obstante, confiamos que los estudiantes y autodidactas que utilicen la obra se conviertan de un modo razonable en acérrimos seguidores y adeptos de C++, al igual que nos ocurre a casi todos los programadores que comenzamos a trabajar con este lenguaje. Así, se tratará de enseñar las técnicas clásicas y avanzadas de programación estructurada, junto con técnicas orientadas a objetos. La programación orientada a objetos no es la panacea universal de programador del siglo XXI, pero le ayudará a realizar tareas que, de otra manera, serían complejas y tediosas.

El libro cuenta con una página Web oficial (`www.mhe.es/joyanes`) donde no sólo podrá consultar y descargarse códigos fuente de los programas del libro, sino también apéndices complementarios y cursos o tutoriales de programación en C++, y de C, especialmente pensados para los lectores sin conocimiento de este lenguaje.

LA EVOLUCIÓN DE C++

El aprendizaje de C++ es una aventura de descubrimientos, en especial, porque el lenguaje se adapta muy bien a diferentes paradigmas de programación, incluyendo entre ellos, *programación orientada a objetos, programación genérica* y la tradicional *programación procedimental* o *estructurada*, tradicional. C++ ha ido evolucionando desde la publicación del libro de Stroustrup *C++. Manual de Referencia con anotaciones* (conocido como ARM y traducido al español por el autor de este libro y por el profesor Miguel Katrib de la Universidad de Cuba) hasta llegar a la actual versión estándar *ISO/ANSI C++ Standard, second edition* (2003) que suelen incorporar casi todos los compiladores y ya muy estabilizada.

Esta edición sigue el estándar ANSI/ISO, aunque en algunos ejemplos y ejercicios, se ha optado por mantener el antiguo estándar, a efectos de compatibilidad, con otras versiones antiguas que pueda utilizar el lector y para aquellos casos en que utilice un compilador no compatible con el estándar, antes mencionado.

Aprenderá muchas características de C++, e incluso las derivadas de C, a destacar:

- Clases y objetos.
- Herencia.
- Polimorfismo y funciones virtuales.
- Sobrecarga de funciones y de operadores.
- Variables referencia.
- Programación genérica, utilizando plantillas (templates) y la Biblioteca Plantillas estándar (**STL,** Standard Template Library).
- Mecanismo de excepciones para manejar condiciones de error.
- Espacio de nombres para gestionar nombres de funciones, clases y variables.

C++ se comenzó a utilizar como un *«C con clases»* y fue a principios de los ochenta cuando comenzó la revolución C++, aunque su primer uso comercial, por parte de una organización de investigación, comenzó en julio de 1983. Como Stroustrup cuenta en el prólogo de la 3.ª edición de su citada obra, C++ nació con la idea de que el autor y sus colegas no tuvieran que programar en ensamblador ni en otros lenguajes al uso (véase Pascal, BASIC, FORTRAN...). La explosión del lenguaje en la comunidad informática hizo inevitable la estandarización, proceso que comenzó en 1987 [Stroustruo 94]. Así nació una primera fuente de estandarización, la ya citada obra : *The Annotated C++ Reference Manual* [**Ellis** 89][1]. En diciembre de 1989 se reunió el comité X3J16 de ANSI, bajo el auspicio de Hewlett-Packard, y en junio de 1991 se realizó el primer esfuerzo de estandarización internacional de la mano de ISO, y así comenzó a nacer el estándar ANSI/ISO C++. En 1995 se publicó un borrador estándar para su examen público y en noviembre de 1997 fue finalmente aprobado el estándar C++ internacional, aunque fue en 1998 cuando el proceso se pudo dar por terminado (**ANSI/ISO C++** *Draft Standard*) conocido como **ISO/IEC 14882: 1998** o simplemente **C++ Estándar** o **ANSI C++**. Stroustrup publicó en 1997 la tercera edición de su libro *The C++ Programming Language* [Stroustrup 97] y en 2000, una actualización que se publicó como *edición especial*[2] *(*traducida por un equipo de profesores de la Facultad de Informática de la Universidad Pontificia de Salamanca en Madrid, dirigidos por el autor de este libro*).* El libro que tiene en sus manos, sigue el estándar ANSI/ISO C++.

COMPILADORES Y COMPILACIÓN DE PROGRAMAS EN C++

Existen numerosos compiladores de C++ en el mercado. Desde ediciones gratuitas y *descargables* a través de Internet hasta profesionales, con costes diferentes, comercializados por diferentes fabricantes. Es difícil dar una recomendación al lector porque casi todos ellos son buenos compiladores, muchos de

[1] Existe versión española de Addison-Wesley Díaz de Santos y traducida por los profesores Manuel Katrib y Luis Joyanes.

[2] Esta obra fue traducida por un equipo de profesores universitarios que dirigió y coordinó el profesor Luis Joyanes, co-autor de esta obra.

ellos con Entornos Integrados de Desarrollo **(EID**). Si usted es estudiante, tal vez la mejor decisión sea utilizar el compilador que le haya propuesto su profesor y que utilice en su Universidad, Instituto Tecnológico o cualquier otro Centro de Formación, donde estudie. Si usted es un lector autodidacta y está aprendiendo por su cuenta existen varias versiones gratuitas que puede descargar desde Internet. Algunos de los más reconocidos son: **Dev-C++** de **Bloodshed** que cumple fielmente el estándar ANSI/ISO C++ (uno de los compiladores utilizado por el autor del libro para editar y compilar los programas incluidos en el mismo) y que corre bajo entornos Windows; **GCC** de GNU que corre bajo los entornos Linux y Unix. Existen muchos otros compiladores gratuitos por lo que tiene donde elegir. Tal vez un consejo más: procure que sea compatible con el estándar ANSI/ISO C++.

Bjarne Stroustrup (creador e inventor de C++) en su página oficial[3] ha publicado el 19 de junio de 2006, *«una lista incompleta de compiladores de C++»*, que le recomiendo lea y visite, aunque el mismo reconoce es imposible tener actualizada la lista de compiladores disponibles. Por su interés incluimos a continuación un breve extracto de su lista recomendada:

Compiladores gratuitos

- Apple C++.
- Bloodshed Dev-C++.
- Borland C++.
- Cygwin (GNU C++).
- Digital Mars C++.
- MINGW - «Minimalist GNU for Windows».
- DJ Delorie's C++ para desarrollo de sistemas DOS/Windows (GNU C++).
- GNU CC fuente.
- Intel C++ para Linux.
- The LLVM Compiler Infrastructure (basado en GCC).
- Microsoft Visual C++ Toolkit 2003.
- Sun Studio.

Compiladores que requieren pago (algunos permiten descargas gratuitas durante periodos de prueba)

- Borland C++.
- Comeau C++ para múltiples plataformas.
- Compaq C++.
- Green Hills C++ para multiples plataformas de sistemas empotrados.
- HP C++.
- IBM C++.
- Intel C++ para Windows, Linux, y sistemas empotrados.
- Interstron C++.
- Mentor Graphics/Microtec Research C++ para sistemas empotrados.
- Microsoft C++.
- Paradigm C++, para sistemas empotrados x86.
- The Portland Group C++.
- SGI C++.
- Sun C++.
- WindRiver's Diab C++ utilizado en muchos sistemas empotrados.

[3] `http://public.research.att.com/~bs/compilers.html`. El artículo *«An incomplete list of C++ compilers»* lo suele modificar Stroustrup y en la cabecera indica la fecha de modificación. En nuestro caso, consultamos dicha página mientras escribíamos el prólogo en la segunda quincena de junio y la fecha de actualización es «19 de junio de 2006».

Stroustrup recomienda un sitio de compiladores gratuitos de C y C++ (**Compilers.net**: www.compilers.net/Dir/Free/Compilers/CCpp.htm).

Nota práctica de compatibilidad C++ estándar

En el artículo antes citado de Stroustrup, recomienda que compile con su compilador, el sencillo programa fuente C++ siguiente. Si usted compila bien este programa, no tendrá problemas con C++ estándar, en caso contrario aconseja buscar otro compilador que sea compatible.

```
#include<iostream>
#include<string>

using namespace std;

int main( )
{
     string s;
     cout << «Por favor introduzca su nombre seguido por Intro \n»;
     cin >> s;
     cout << «Hola, « << s << '\n';
     return 0; // esta sentencia return no es necesaria
}
```

Nota práctica sobre compilación de programas fuente con el compilador Dev C++ y otros compiladores compatibles ANSI/ISO C++

Muchos programas del libro han sido compilados y ejecutados en el compilador de *freeware* (de libre distribución) `BloodShed` **`DEV-C++`**[4], versión 4.9.9.2. Dev C++ es un editor de múltiples ventanas integrado con un compilador de fácil uso que permite la compilación, enlace y ejecución de programas C o C++.

Las sentencia **`system(«PAUSE»);`** y **`return EXIT_SUCCES;`** son incluidas por defecto por el entorno **Dev** en todos los programas escritos en C++. A efectos de compatibilidad práctica, si usted no utiliza el compilador Dev C++ o no desea que en sus listados aparezca estas sentencias, puede sustituirlas bien por otras equivalentes o bien quitar las dos y sustituirlas por **`return 0,`** como recomienda Stroustrup para terminar sus programas y que es el método habitualmente empleado en el libro.

`system(«PAUSE»);` detiene la ejecución del programa hasta que se pulsa una tecla. Las dos sentencias siguientes de C++ producen el mismo efecto:

```
cout << «Presione ENTER para terminar»;
cint.get();
```

`return EXIT_SUCCES;` devuelve el estado de terminación correcta del programa al sistema operativo. La siguiente sentencia de C++ estándar, produce el mismo efecto:

```
return 0;
```

[4] `http://www.bloodshed.net/`.

Si en el compilador con el cual trabaja no le funciona alguna de las sentencias anteriores puede sustituirlas por las indicadas en la Tabla P.1.

Tabla P.1. Dev C++ *versus* otros compiladores de C++ (en compilación)

Sustitución de sentencias	
DEV C++	**C++**
`system(«PAUSE»);`	`cout << «Presione ENTER para terminar»; cint.get();`
`return EXIT_SUCCES;`	`return 0;`

OBJETIVOS DEL LIBRO

C++ es un superconjunto de C y su mejor extensión. Este es un tópico conocido por toda la comunidad de programadores del mundo. Cabe preguntarse como hacen muchos autores, profesores, alumnos y profesionales: ¿se debe aprender primero C y luego C++? Stroustrup y una gran mayoría de programadores contestan así: No sólo es innecesario aprender primero C, sino que además es una mala idea. Nosotros no somos tan radicales y pensamos que se puede llegar a C++ procediendo de ambos caminos, aunque es lógico la consideración citada anteriormente, ya que efectivamente los hábitos de programación estructurada de C pueden retrasar la adquisición de los conceptos clave de C++, pero también es cierto que en muchas casos ayuda considerablemente en el aprendizaje.

Este libro supone que el lector no es programador de C, ni de ningún otro lenguaje, aunque también somos conscientes de que el lector que haya seguido un primer curso de programación en algoritmos o en algún lenguaje estructurado, llámese Pascal o cualquier otro, éste le ayudará favorablemente al correcto y rápido aprendizaje de la programación en C++ y obtendrá el máximo rendimiento de esta obra. Sin embargo, si ya conoce C, naturalmente no tendrá ningún problema en su aprendizaje, muy al contrario, bastará que lea con detalle las diferencias esenciales de los primeros capítulos (en caso de duda puede también consultar la página oficial del libro donde podrá encontrar numerosa documentación sobre C), de modo que irá integrando gradualmente los nuevos conceptos que irá encontrando a medida que avance en la obra con los conceptos clásicos de C. El libro pretende enseñar a programar utilizando tres conceptos fundamentales:

1. *Algoritmos* (conjunto de instrucciones programadas para resolver una tarea específica).
2. *Datos y Estructuras de Datos* (una colección de datos que se proporcionan a los algoritmos que se han de ejecutar para encontrar una solución: los datos se organizarán en *estructuras de datos).*
3. *Objetos* (el conjunto de datos y algoritmos que los manipulan, encapsulados en un tipo de dato nuevo conocido como objeto).

Los dos primeros aspectos, **algoritmos** y **datos**, han permanecido invariables a lo largo de la corta historia de la informática/computación, pero la interrelación entre ellos sí que ha variado y continuará haciéndolo. Esta interrelación se conoce como *paradigma de programación.*

En el paradigma de programación procedimental (*procedural* o por procedimientos) un problema se modela directamente mediante un conjunto de algoritmos. Por ejemplo, la nómina de una empresa o la gestión de ventas de un almacén, se representa como una serie de procedimientos que manipulan datos. Los datos se almacenan separadamente y se accede a ellos o bien mediante una posición global o mediante parámetros en los procedimientos. Tres lenguajes de programación clásicos, FORTRAN, Pascal y C, han representado el arquetipo de la programación *procedimental*, también relacionada estrechamente y, a veces, conocida como *programación estructurada*. La programación con soporte en C++, propor-

ciona el paradigma *procedimental c*on un énfasis en funciones, plantillas de funciones y algoritmos genéricos.

En la década de los ochenta, el enfoque del diseño de programas se desplazó desde el paradigma *procedimental* al orientado a objetos apoyado en los tipos abstractos de datos (TAD). En este paradigma se modela un conjunto de abstracciones de datos. En C++ estas abstracciones se conocen como **clases**. Las clases contienen un conjunto de instancias o ejemplares de la misma que se denominan objetos, de modo que un programa actúa como un conjunto de objetos que se relacionan entre sí. La gran diferencia entre ambos paradigmas reside en el hecho de que los algoritmos asociados con cada clase se conocen como *interfaz pública* de la clase y los datos se almacenan privadamente dentro de cada objeto de modo que el acceso a los datos está oculto al programa general y se gestionan a través de la interfaz.

C++ es un lenguaje *multiparadigma*. Soporta ambos tipos de paradigmas. La gran ventaja es que se puede proporcionar la solución que mejor; resuelva cada problema con el paradigma más adecuado, dado que ningún paradigma resuelve definitivamente todos los problemas. El inconveniente es que el lenguaje se vuelve más grande y complejo, pero este inconveniente está quedando prácticamente resuelto por el citado proceso de estandarización, internacional a que ha sido sometido C++, lo que ha conseguido que esa implícita dificultad se haya convertido en facilidad de uso a medida que se controla y domina el lenguaje. También los fabricantes de compiladores han contribuido a ello y al núcleo fundamental del lenguaje le han añadido una serie de bibliotecas de funciones y de plantillas (tal como **STL**) que han simplificado notablemente las tareas de programación y han facilitado de sobremanera que éstas sean muy eficientes.

Así pues, en resumen, los objetivos fundamentales de esta obra son: *introducción a la programación estructurada*, *estructuras de datos* y *programación orientada a objetos* con el lenguaje estándar **ANSI/ ISO C++.**

EL LIBRO COMO HERRAMIENTA DOCENTE

La experiencia del autor desde hace muchos años con obras muy implantadas en el mundo universitario como *Programación en C++* (1.ª edición), *Fundamentos de programación* (en su 3.ª edición), *Progamación en C* (en su 2.ª edición), *Programación en Pascal* (en su 4.ª edición), y *Programación en BASIC* (que alcanzó tres ediciones y numerosísimas reimpresiones en la década de los ochenta) nos ha llevado a mantener la estructura de esta obra, actualizándola a los contenidos que se prevén para los estudiantes del actual siglo XXI. Por ello en el contenido de la obra hemos tenido en cuenta no sólo las directrices de los planes de estudio españoles de ingeniería informática e ingeniería técnica en informática de sistemas y de gestión, y licenciaturas en ciencias de la computación, matemáticas, físicas..., sino también de ingenierías tales como industriales, telecomunicaciones, agrónomos o geodesia. Nuestro conocimiento del mundo educativo latinoamericano nos ha llevado a pensar también en las carreras de ingeniería de sistemas computacionales y las licenciaturas en informática y en sistemas de información, como se las conoce en aquel continente americano.

Por todo lo anterior, el contenido del libro intenta seguir un programa estándar de un primer curso de introducción a la programación y, según situaciones, un segundo curso de programación de nivel medio, en asignaturas tales como *Metodología de la programación, Fundamentos de programación, Introducción a la programación...* Asimismo, se ha buscado seguir las directrices emanadas de la ACM para curricula actuales y las vigentes en universidades latinoamericanas, muchas de las cuales conocemos y con las que tenemos relaciones profesionales.

El contenido del libro abarca los citados programas y comienza con la introducción a la computación y a la programación, para llegar a estructuras de datos y objetos. Por esta circunstancia la estructura del curso no ha de ser secuencial en su totalidad sino que el profesor/maestro y el alumno/lector podrán estudiar sus materias en el orden que consideren más oportuno. Esta es la razón principal por la cual el libro se ha organizado en cuatro partes con numerosos apéndices incluidos en la página *web* oficial del mismo, con el objeto de que el lector seleccione y se «baje» aquellos apéndices que considere de su interés, y de este modo no incrementar el número de páginas de la obra.

Se trata de describir los dos paradigmas más populares en el mundo de la programación: el *procedimental* y el *orientado a objetos*. Los cursos de programación en sus niveles inicial y medio están evolucionando para aprovechar las ventajas de nuevas y futuras tendencias en ingeniería de software y en diseño de lenguajes de programación, específicamente diseño y programación orientada a objetos. Algunas facultades y escuelas de ingenieros, junto con la nueva formación profesional (ciclos formativos de nivel superior) en España y en Latinoamérica, han introducido a sus alumnos en la programación orientada a objetos, inmediatamente después del conocimiento de la programación estructurada, e incluso —en ocasiones— antes o en paralelo. Por esta razón, una metodología que se podría seguir sería impartir un curso de fundamentos de programación seguido de estructuras de datos y luego seguir con un segundo nivel de programación avanzada y programación orientada a objetos que constituyen las cuatro partes del libro. Pensando en aquellos alumnos que necesiten profundizar en los temas tratados en el capítulo 1, se han escrito los Apéndices 1 y 2 y «subido» a la página web del libro con una ampliación de Introducción a las computadoras y una Introducción a la Programación con el lenguaje algorítmico, pseudocódigo, y ejemplos en C y C++.

CARACTERÍSTICAS IMPORTANTES DEL LIBRO

Programación en C++ utiliza los siguientes elementos clave para conseguir obtener el mayor rendimiento del material incluido en sus diferentes capítulos:

Contenido. Enumera los apartados descritos en el capítulo.

Introducción. Abre el capítulo con una breve revisión de los puntos y objetivos más importantes que se tratarán y todo aquello que se puede esperar del mismo.

Conceptos clave. Enumera los términos informáticos y de programación más notables que se tratarán en el capítulo.

Descripción del capítulo. Explicación usual de los apartados correspondientes del capítulo. En cada capítulo se incluyen ejemplos y ejercicios resueltos. Los listados de los programas completos o parciales se escriben en letra Courier con la finalidad principal de que puedan ser identificados fácilmente por el lector.

Resumen del capítulo. Revisa los temas importantes que los estudiantes y lectores deben comprender y recordar. Busca también ayudar a reforzar los conceptos clave que se han aprendido en el capítulo.

Ejercicios. Al final de cada capítulo se proporciona a los lectores una lista de ejercicios sencillos de modo que le sirvan de oportunidad para que puedan medir el avance experimentado mientras leen y siguen —en su caso— las explicaciones del profesor relativas al capítulo.

Problemas. Después del apartado Ejercicios, se añaden una serie de actividades y proyectos de programación que se le proponen al lector como tarea complementaria de los ejercicios y de un nivel de dificultad algo mayor.

Ejercicios resueltos y problemas resueltos. En estas secciones, el lector encontrará enunciados de ejercicios y problemas complementarios, cuyas soluciones encontrará en las dos fuentes siguientes:

1. ***Programación en C++: Un enfoque práctico.*** Madrid: McGraw-Hill, Colección *Schaum*, 2006, de Luis Joyanes y Lucas Sánchez.
2. Portal del libro: `www.mhe.es/joyanes`.

El libro *Programación en C++: Un enfoque práctico*, perteneciente a la prestigiosa Colección *Schaum*, de libros prácticos, ha sido escrito por el autor del libro y el profesor Lucas Sánchez, como un libro eminentemente práctico y complementario de éste que tiene usted en sus manos, por lo cual con-

tiene un gran número de ejercicios y problemas propuestos y resueltos. Los códigos fuente de los *Ejercicios resueltos* y *Problemas resueltos* de los 20 primeros capítulos de este libro los encontrará usted en las dos fuentes citadas anteriormente. De este modo, el lector que lo desee podrá verificar y comprobar su propia solución con la solución planteada en el citado libro de la colección Shaum o bien en la página web del libro.

Recuadro. Conceptos importantes que el lector debe considerar durante el desarrollo del capítulo.

Consejo. Ideas, sugerencias, recomendaciones... al lector, con el objetivo de obtener el mayor rendimiento posible del lenguaje y de la programación.

Precaución. Advertencia al lector para que tenga cuidado al hacer uso de los conceptos incluidos en el recuadro adjunto.

Reglas. Normas o ideas que el lector debe seguir preferentemente en el diseño y construcción de sus programas.

ORGANIZACIÓN DEL LIBRO

El libro se divide en cuatro partes que unidas constituyen un curso completo de programación en C++. Dado que el conocimiento es acumulativo, los primeros capítulos proporcionan el fundamento conceptual para la comprensión y aprendizaje de C++ y una guía a los estudiantes a través de ejemplos y ejercicios sencillos, y los capítulos posteriores presentan de modo progresivo la programación en C++ en detalle, tanto en el paradigma procedimental como en el orientado a objetos. Los apéndices contienen un conjunto de temas importantes que incluyen desde guías de sintaxis de ANSI/ISO C++ hasta un glosario de términos o una biblioteca de funciones y clases, junto con una extensa bibliografía de algoritmos, estructura de datos, programación orientada a objetos y una amplia lista de sitios de Internet (URL) donde el lector podrá complementar, ampliar y profundizar en el mundo de la programación y en la introducción a la ingeniería de software.

Parte I. Fundamentos de programación en C++

Esta parte es un primer curso de programación para alumnos principiantes en asignaturas de introducción a la programación . Esta parte sirve tanto para cursos de C++ como de C (en este caso con la ayuda de los Apéndices A y B). Esta parte comienza con una introducción a la informática y a las ciencias de la computación como a la programación. Describe los elementos básicos constitutivos de un programa y las herramientas de programación utilizadas tales como algoritmos, diagramas de flujo, etc. Asimismo se incluye un curso del lenguaje C++ y técnicas de programación que deberá emplear el lector en su aprendizaje de programación.

Capítulo 1. ***Introducción a la ciencia de la computación y a la programación.*** Proporciona una revisión de las características más importantes necesarias para seguir bien un curso de programación básico y avanzado en C++. Para ello se describe la organización física de una computadora junto con el concepto de algoritmo y los métodos más eficientes para la resolución de problemas con computadora. Se explican también los diferentes tipos de programación y una breve historia del lenguaje C++.

Capítulo 2. ***El lenguaje C++. Elementos básicos.*** Enseña los fundamentos de C++, organización y estructura general de un programa, función `main ()`, ejecución, depuración y prueba de un programa, elementos de un programa, tipos de datos (el tipo de dato bool), constantes, variables y entradas/salidas de datos (`cin` y `cout`).

Capítulo 3. ***Operadores y expresiones.*** Se describen los conceptos y tipos de operadores y expresiones, conversiones y precedencias. Se destacan operadores especiales tales como manipulación

de bits, condicional, `sizeof`, `()`, `[]`, `::`, coma, etc., y se analizan los conceptos de conversión de tipos, prioridad y asociatividad entre operadores.

Capítulo 4. Estructuras de selección: sentencias `if` y `switch`. Introduce al concepto de estructura de control y, en particular, estructuras de selección, tales como `if`, `if-else`, `case` y `switch`. Expresiones condicionales con el operador `?:`, evaluación en cortocircuito de expresiones lógicas,.errores frecuentes de programación y puesta a punto de programas.

Capítulo 5. Estructuras repetitivas: bucles (`for`, `while` y `do-while`). El capítulo introduce las estructuras repetitivas (`for`, `while` y `do-while`). Examina la repetición (iteración) de sentencias en detalle y compara los bucles controlados por centinela, bandera, etc. Explica precauciones y reglas de uso de diseño de bucles. Compara los tres diferentes tipos de bucles, así como el concepto de bucles anidados.

Capítulo 6. Funciones. Examina el diseño y construcción de módulos de programas mediante funciones. Se define la estructura de una función, prototipos y parámetros. El concepto de funciones en línea (*inline*), uso de bibliotecas de funciones, clases de almacenamiento, ámbitos, visibilidad de una función. Asimismo se introduce el concepto de recursividad y plantillas de funciones.

Capítulo 7. Arrays/ Arreglos (listas y tablas). Examina la estructuración de los datos en *arrays* (*arreglos*) o grupos de elementos dato del mismo tipo. El capítulo presenta numerosos ejemplos de arrays de uno, dos o múltiples índices. Se realiza una introducción a los algoritmos de ordenación y búsqueda de elementos en una lista.

Capítulo 8. Estructuras y uniones. Conceptos de estructuras, declaración, definición, iniciación, uso y tamaño. Acceso a estructuras. *Arrays* (arreglos) de estructuras y estructuras anidadas. Uniones y enumeraciones.

Capítulo 9. Punteros (apuntadores). Presenta una de las características más potentes y eficientes del lenguaje C++, los punteros. Este capítulo proporciona explicación detallada de los punteros, *arrays* de punteros, punteros de cadena, aritmética de punteros, punteros constantes, punteros como argumentos de funciones, punteros a funciones y a estructuras.

Capítulo 10. Asignación dinámica de memoria. En este capítulo se describe la gestión dinámica de la memoria junto con los operadores `new` y `delete`. Se dan reglas de funcionamiento de ambos operadores y se describe el concepto de arrays dinámicos y asignación de memoria para *arrays.* Se examinan los tipos de memoria en C++ así como los modos de asignación y liberación de memoria.

Capítulo 11. Cadenas. Se examina el concepto de cadena (`string`) así como las relaciones entre punteros, *arrays* y cadenas en C++. Se introducen conceptos básicos de manipulación de cadenas junto con operaciones básicas tales como longitud, concatenación, comparación, conversión y búsqueda de caracteres y cadenas.

Capítulo 12. Ordenación y búsqueda. Dos de las operaciones más importantes que se realizan en algoritmos y programas de cualquier entidad y complejidad son: ordenación de elementos de una lista o tabla y búsqueda de un elemento en dichas listas o tablas. Los métodos básicos más usuales de búsqueda y ordenación se describen en el capítulo. Asimismo, se explica el concepto de análisis de algoritmos de ordenación y búsqueda.

Parte II. Programación orientada a objetos

En esta parte se describen las propiedades fundamentales de la programación orientada a objetos y los métodos de implementación de las mismas con el lenguaje C++. Así, se estudian y analizan: *clases y objetos, clases derivadas y herencia, polimorfismo* y la *genericidad* (*plantillas*, templates), soporte también del paradigma de programación genérica. Los tres capítulos que constituyen esta parte, se han diseñado como una introducción a la programación orientada a objetos; los Capítulos 21 y 22 que tratan sobre «*sobrecarga de operadores*» *y* «*excepciones*» están directamente relacionados con los tres capítulos de esta parte, ya que también son propiedades características del citado paradigma de programación.

Capítulo 13. Clases y objetos. Este capítulo muestra la forma de implementar la abstracción de datos, mediante tipos abstractos de datos, clases. Se describen los conceptos de clase y objeto, así como el sistema para definición de una clase. El capítulo examina el método de inicialización de objetos, junto con el concepto de constructores y destructores de una clase.

Capítulo 14. Clases derivadas: herencia y polimorfismo. Dos de las propiedades más importantes de la programación orientada a objetos son: *herencia* y *polimorfismo*. La herencia es un sistema de *reusabilidad* de software en el que nuevas clases se desarrollan rápida y fácilmente a partir de las clases existentes y añadiendo nuevas propiedades a las mismas. Este capítulo examina las nociones de clases base y clases derivadas, herencia *protegida*, *pública* y *privada* (`protected`, `public`, `private`), constructores y destructores en clases base y derivadas. Se describen los diferentes tipos de herencia: simple y múltiple. El polimorfismo y la ligadura dinámica se describen también a lo largo del capítulo.

Capítulo 15. Genericidad: plantillas (templales). Examina una de las incorporaciones más importantes de la evolución del lenguaje C++: las plantillas (*templates*). Se describen el concepto y la definición de las plantillas de funciones. Las plantillas de clases denominadas tipos *parametrizados* permiten definir tipos genéricos tales como una cola de enteros, una cola de reales (`float`), una cola de cadenas, etc. Se presenta una aplicación práctica y se realiza una comparativa de las plantillas y el polimorfismo.

Parte III. Estructuras de datos

Capítulo 16. Flujos y archivos: biblioteca estándar E/S. Contiene una descripción abierta del tratamiento del nuevo estilo orientado a objetos de entrada/salida en C++. Este capítulo examina las diversas características de E/S de C++ que incluye la salida con el operador de inserción de flujos, la entrada con el operador de extracción de flujos, E/S con seguridad de tipos. Se analizan el uso y aplicación de los manipuladores e indicadores de formato. El concepto de archivo junto con su definición e implementación es motivo de estudio en este capítulo. Las operaciones usuales se estudian con detenimiento

Capítulo 17. Listas enlazadas. Una lista enlazada es una estructura de datos que mantiene una colección de elementos, pero el número de ellos no se conoce por anticipado o varía en un amplio rango. La lista enlazada se compone de elementos que contienen un valor y un puntero. El capítulo describe los fundamentos teóricos y las operaciones que se pueden realizar en la lista enlazada. También se describen los distintos tipos de listas enlazadas.

Capítulo 18. Pilas y colas. Las ideas abstractas de pila y cola se describen en el capítulo. Pilas y colas se pueden implementar de diferentes maneras, bien con vectores (arrays) o con listas enlazadas.

Capítulo 19. Recursividad. El importante concepto de recursividad (propiedad de una función de llamarse a sí misma) se introduce en el capítulo junto con algoritmos complejos de ordenación y búsqueda en los que además se estudia su eficiencia. Entre los algoritmos importantes se explica el método de ordenación rápida *QuickSort*.

Capítulo 20. Árboles. Los árboles son otro tipo de estructura de datos dinámica y no lineal. Las operaciones básicas en los árboles junto con sus operaciones fundamentales se estudian en el capítulo. Se describen también los árboles binarios y árboles binarios de búsqueda como elementos clave en el diseño y construcción de estructura de datos complejas.

Parte IV. Programación avanzada

En esta parte del libro se describen dos características importantes de C++ -también existen en otros lenguajes de programación orientados a objetos como Java y Ada- que se pueden considerar como propiedades de orientación a objetos o como propiedades complementarias, pero que son de gran uso en

programación avanzada y cuya correcta aplicación por el programador le permitirá diseñar y construir programas muy eficientes.

Capítulo 21. Sobrecarga de operadores. Es una de las características más populares de cualquier curso de programación en C++. La sobrecarga de operadores permite al programador indicar al compilador cómo utilizar operadores existentes con objetos de tipos nuevos. C++ utiliza la sobrecarga con tipos incorporados tales como enteros, reales y carácter. Un ejemplo típico es la concatenación de cadenas mediante el operador «+» que une una cadena a continuación de otra.

Capítulo 22 Excepciones. Se examina en este capítulo una de las mejoras más sobresalientes del lenguaje C++. El manejo de excepciones permite al programador escribir programas que sean más robustos, más tolerantes a fallos y más adecuados a entornos de misión y negocios críticos. El capítulo introduce a los fundamentos básicos del manejo de excepciones con bloques `try`, sentencias throw y bloques `catch`; indica cómo y cuándo se vuelve a lanzar una excepción y se incluyen aplicaciones prácticas de manejo de excepciones.

APÉNDICES INCLUIDOS EN LA WEB (página oficial del libro: `www.mhe.es/joyanes`)

En todos los libros dedicados a la enseñanza y aprendizaje de técnicas de programación es frecuente incluir apéndices de temas complementarios a los explicados en los capítulos anteriores. Estos apéndices sirven de guía y referencia de elementos importantes del lenguaje y de la programación de computadoras. En esta edición se ha optado por colocarlos en la Web, de modo que el lector pueda *«bajarlos»* de ella cuando lo considere oportuno y en función de su lectura y aprendizaje.

Apéndice A. C++ frente a C. Estudio extenso de comparación de características de los lenguajes C y C++ cuyo objetivo es facilitar la migración de un lenguaje a otro con facilidad.
Apéndice B. Guía de sintaxis de ANSI/ISO C++. Descripción detallada de los elementos fundamentales del estándar C++.
Apéndice C. Operadores (prioridad). Tabla que contiene todos los operadores y el orden de prioridad en las operaciones cuando aparecen en expresiones.
Apéndice D. Código de caracteres ASCII. Listado del juego de caracteres del código ASCII utilizado en la actualidad en la mayoría de las computadoras.
Apéndice E. Listado por orden alfabético de las palabras reservadas en ANSI/ISO C++, al estilo del diccionario. Definición y uso de cada palabra reservada, con ejemplos sencillos de aplicación.
Apéndice F. Biblioteca defunciones estándar ANSI/ISO C++. Diccionario alfabético de las funciones estándar de la biblioteca estándar de ANSI/ISO C++, con indicación de la sintaxis del prototipo de cada función. Una descripción de su misión, junto con algunos ejemplos sencillos de la misma.
Apéndice G. Biblioteca de clases de ANSI/ISO C++. Diccionario de clases correspondientes a la biblioteca de clases estándar de C++.
Apéndice H. Glosario. Minidiccionario de términos importantes de programación en inglés, con su traducción al español y una breve descripción de los mismos.
Apéndice I. Recursos (Libros / Revistas / URLs de la Web sobre C++). Colección de recursos Web de interes para el lector, tanto para su etapa de aprendizaje de la programación en C++, como para su etapa profesional de programador.
Apéndice J. Bibliografía. Enumeración de los libros más sobresalientes empleados por el autor en la escritura de esta obra, así como otras obras importantes complementarias que ayuden al lector que desee profundizar o ampliar aquellos conceptos que considere necesario conocer con más detenimiento.

Pensando en los lectores que deseen profundizar en los conceptos introductorias explicados en el Capítulo 1 *«Introducción a la computación y a la programación»* se han colocado en el portal web del

libro dos apéndices complementarios a modo de pequeños cursos teórico-práctico de *introducción a las computadoras, algoritmos* y *programación,* con la ayuda de un lenguaje algoritmico o pseudocógio.

Apéndice 1. *Introducción a las computadoras y a los lenguajes de programación*.
Apéndice 2. *Metodología de la programación y desarrollo de software*.

AGRADECIMIENTOS EN LA PRIMERA EDICIÓN

Un libro nunca es fruto único del autor, sobre todo si el libro está concebido como libro de texto y autoaprendizaje, y pretende llegar a lectores y estudiantes de informática y de computación, y, en general, de ciencias e ingeniería, así como a autodidactas en asignaturas tales como programación (introducción, fundamentos, avanzada, etc.). Esta obra no es una excepción a la regla y son muchas las personas que me han ayudado a terminarla. En primer lugar, mis colegas de la Universidad Pontificia de Salamanca en el campus de Madrid, y en particular del Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software de la misma, que desde hace muchos años me ayudan y colaboran en la impartición de las diferentes asignaturas del departamento y, sobre todo, en la elaboración de los programas y planes de estudio de las mismas. A todos ellos les agradezco públicamente su apoyo y ayuda.

En especial deseo agradecer la revisión, comentarios, ideas, consejos y sugerencias, que sobre este libro, en particular, me han dado los siguientes profesores de universidades españolas:

Ignacio Zahonero Martínez, profesor de la Facultad de Informática de la Universidad Pontificia de Salamanca en el campus de Madrid.

María Luisa Diez Platas, profesora de la Facultad de Informática de la Universidad Pontificia de Salamanca en el campus de Madrid.

Eduardo González Joyanes, profesor de la Facultad de Ingeniería de la Universidad Católica de Ávila.

También he de agradecer a otros profesores su cuidada revisión de algunos capítulos de esta obra, así como la aportación de algunos gráficos y dibujos:

Sergio Ríos Aguilar, profesor de la Facultad de Informática de la Universidad Pontificia de Salamanca en el campus de Madrid.

Francisco Mata Mata, profesor de la Escuela Politécnica Superior de la Universidad de Jaén.

Luis Rodríguez Baena, profesor de la Facultad de Informática de la Universidad Pontificia de Salamanca en el campus de Madrid.

A los restantes profesores del grupo de Tecnologías Orientadas a Objetos de nuestra Facultad que ponen en marcha todas las asignaturas relacionadas con estas tecnologías: Héctor Castan Rodríguez, Salvador Sánchez Sánchez, Paloma Centenera Centenera, Francisco Morales Arcía, Rosa Hernández Prieto, Miguel Ángel Sicilia Urbán, María Borrego, Víctor Martín García, Rafael Ojeda Martín, Andrés Castillo y Antonio Reus Hungría.

Muchos otros profesores han ayudado en la concepción y realización de esta obra, de una u otra manera, apoyando con su continua colaboración y sugerencia de ideas la puesta en marcha de asignaturas del área de programación; con el riesgo de olvidar a alguno y pidiéndole, por anticipado, mi involuntaria omisión, he de citar de modo especial a los siguientes compañeros —y sin embargo amigos— en el Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software de la Facultad de Informática y Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca en el campus de Madrid: Isabel Torralvo, Mercedes Vargas, Luis Villar, Óscar San Juan, Yago Sáez, Antonio Muñoz, Mónica Vázquez, Angela Carrasco, Matilde Fernández, Juan Antonio Riesco, Joaquín Aveger, Lucas Parra y Miguel Sánchez

Además de estos compañeros en docencia, no puedo dejar de agradecer, una vez más, a mi editora —y sin embargo amiga— **Concha Fernández Madrid**, las constantes muestras de afecto y comprensión que siempre tiene, y ésta no ha sido una excepción, hacia mi persona y mi obra. Sus continuos consejos,

sugerencias y recomendaciones, siempre son acertados y, además, fáciles de seguir; por si no fuera suficiente, siempre benefician a la obra.

Mi eterno agradecimiento a todas las personas anteriores. Naturalmente, no puedo dejar de agradecer a mis numerosos alumnos, estudiantes y lectores, en general, españoles y latinoamericanos, que, continuamente, me aconsejan, critican y me proporcionan ideas para mejoras continuas a mis libros. Sin todo lo que he aprendido, sigo aprendiendo y seguiré aprendiendo de ellos y su aliento continuo, sería prácticamente imposible para mí terminar nuevas obras y, en especial, este libro. También deseo expresar mi agradecimiento a tantos y tantos colegas de universidades españolas y latinoamericanas que apoyan mi labor docente y editorial.

Mi reconocimiento y agradecimiento eterno a todos: alumnos, lectores, colegas, profesores, maestros, monitores y editores. Gracias por vuestra ayuda.

En Carchelejo, Jaén (Andalucía), verano de 1999.

AGRADECIMIENTOS EN LA SEGUNDA EDICIÓN

A mis compañeros del Departamento de Lenguajes y Sistemas Informáticos de la Facultad de Informática y Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca en el *campus* de Madrid, que revisaron las primeras pruebas de esta obra y me dieron consejos sobre las mismas:

- **Ignacio Zahonero Martínez**.
- **Lucas Sánchez García**.
- **Matilde Fernández Azuela**.

Gracias, compañeros, y sin embargo, amigos.

A mi editor y amigo, **Carmelo Sánchez**, que una vez más, me ha aconsejado, me ha revisado y me ha acompañado con sus buenas artes, costumbres y su profesionalidad, en todo el camino que ha durado la edición de este libro.

De un modo muy especial y con mi agradecimiento eterno, a mis lectores, a los estudiantes de España y de Latinoamérica, que han estudiado o consultado la primera edición de esta obra, a mis colegas —profesores y maestros— de España y Latinoamérica que han tenido la amabilidad de consultar o seguir su contenido en sus clases y a todos aquellos que me han dado consejos, sugerencias, propuestas y revisiones. Mi reconocimiento más sincero y de nuevo «mi agradecimiento eterno»; son el aliento diario que me permite continuar con esta hermosa tarea que es la comunicación con todos ellos. Gracias.

En Carchelejo (Sierra Mágina), Andalucía (España), julio de 2006.

P A R T E   I

FUNDAMENTOS DE PROGRAMACIÓN

CAPÍTULO 1

Introducción a la ciencia de la computación y a la programación

Contenido

INTRODUCCIÓN

Las computadoras electrónicas modernas son uno de los productos más importantes de los siglos XX y XXI Y especialmente la actual década. Son una herramienta esencial en muchas áreas: industria, gobierno, ciencia, educación..., en realidad en casi todos los campos de nuestras vidas. El papel de los programas de computadoras es esencial; sin una lista de instrucciones a seguir, la computadora es virtualmente inútil. Los lenguajes de programación nos permiten escribir esos programas y por consiguiente comunicarnos con las computadoras.

En esta obra, usted comenzará a estudiar la ciencia de la computación o informática a través de uno de los lenguajes de programación más versátiles disponibles hoy día: el lenguaje C++. Este capítulo le introduce a la computadora y sus componentes, así como a los lenguajes de programación, y a la metodología a seguir para la resolución de problemas con computadoras y con una herramienta denominada C++.

En el capítulo se describirá el concepto y organización física (*hardware*) y lógica (*software*) de una computadora junto con las formas diferentes de representación de la información. El concepto de algoritmo como herramienta de resolución de problemas es otro de los temas que se abordan en el capítulo.

Las dos paradigmas más populares y que soporta el lenguaje de programación C++ son: *programación estructurada* y *programación orientada a objetos*. Junto con las características de los diferentes tipos de software —en particular el sistema operativo— y de los lenguajes de programación y, en particular, C++ y UML 2.0 se articula la segunda parte del contenido del capítulo.

CONCEPTOS CLAVE

- Algoritmo.
- CD-ROM, CDR/W.
- Compilador.
- Computadora.
- Diagrama de flujo.
- Diagrama N-S.
- Disquete.
- DVD.
- DVD alta definición.
- *Hardware*.
- Intérprete.
- Lenguaje de máquina.
- Lenguaje de programación.
- Lenguaje ensamblador.
- Memoria.
- Memoria auxiliar.
- Memoria central.
- Microprocesador.
- Módem.
- *Software*.
- Unidad central de proceso.

1.1. ¿QUÉ ES UNA COMPUTADORA?

Una **computadora**[1] es un dispositivo electrónico utilizado para procesar información y obtener resultados. Los datos y la información se pueden introducir en la computadora por la **entrada** (*input*) y a continuación se procesan para producir una **salida** (*output*, resultados), como se observa en la Figura 1.1. La computadora se puede considerar como una unidad en la que se ponen ciertos datos, *entrada de datos*, procesa estos datos y produce unos *datos de salida*. Los datos de entrada y los datos de salida pueden ser realmente cualquier cosa, texto, dibujos o sonido. El sistema más sencillo de comunicarse una persona con la computadora es esencialmente mediante un ratón (*mouse*), un teclado y una pantalla (monitor). Hoy día existen otros dispositivos muy populares tales como escáneres, micrófonos, altavoces, cámaras de vídeo, cámaras digitales, etc.; de igual manera, mediante *módems*, es posible conectar su computadora con otras computadoras a través de redes, siendo la más importante, la red **Internet**.

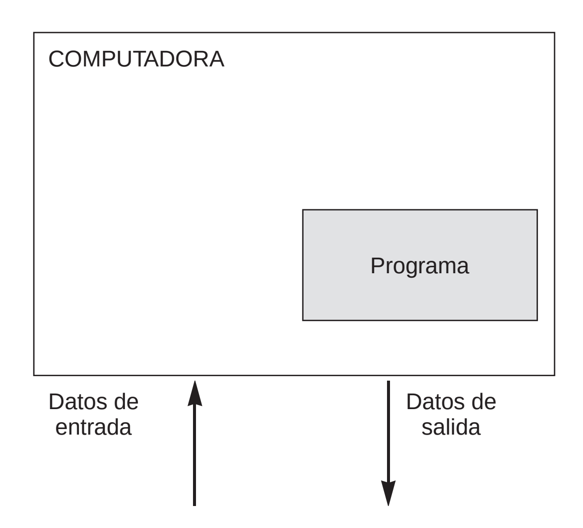


Figura 1.1. Proceso de información en una computadora.

Los componentes físicos que constituyen la computadora, junto con los dispositivos que realizan las tareas de entrada y salida, se conocen con el término ***hardware***. El conjunto de instrucciones que hacen funcionar a la computadora se denomina ***programa***, que se encuentra almacenado en su memoria; a la persona que escribe programas se llama ***programador*** y al conjunto de programas escritos para una computadora se llama ***software***. Este libro se dedicará casi exclusivamente al *software*, pero se hará una breve revisión del *hardware* como recordatorio o introducción según sean los conocimientos del lector en esta materia. En el Anexo A de la página oficial del libro (`www.mhe.es/joyanes`) puede encontrar una amplia información de "Introducción a las computadoras", si desea ampliar este apartado.

1.2. ORGANIZACIÓN FÍSICA DE UNA COMPUTADORA (HARDWARE)

La mayoría de las computadoras, grandes o pequeñas, están organizadas como se muestra en la Figura 1.2. Constan fundamentalmente de tres componentes principales: *Unidad Central de Proceso* (**UCP**) o *procesador* (compuesta de la **UAL**, **U**nidad **A**ritmética y **L**ógica, y la **UC**, **U**nidad de **C**ontrol); la *memoria principal* o *central* y el *programa*.

[1] En España está muy extendido el término **ordenador** para referirse a la traducción de la palabra inglesa *computer*.

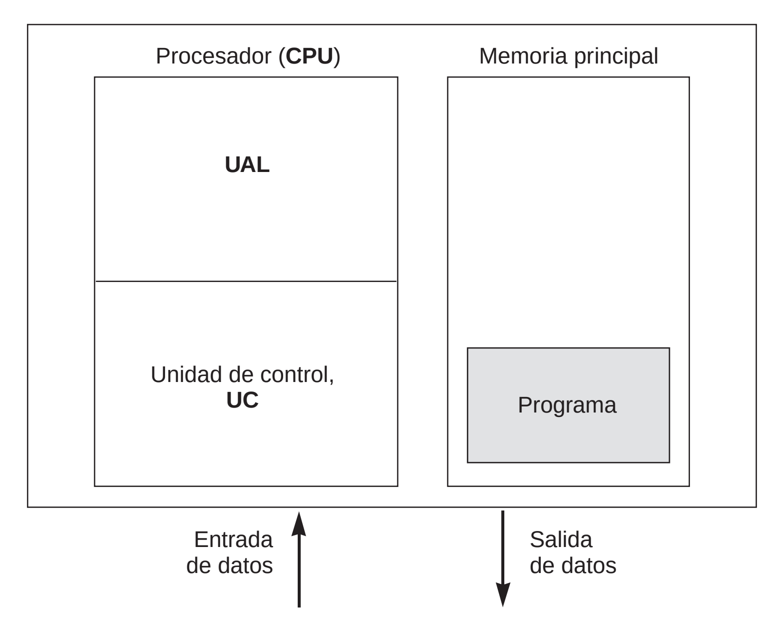


Figura 1.2. Organización física de una computadora.

Si a la organización física de la Figura 1.2 se le añaden los dispositivos para comunicación con la computadora, aparece la estructura típica de un sistema de computadora: dispositivos de entrada, dispositivos de salida, memoria externa y el procesador/memoria central con su programa (Fig.1.3).

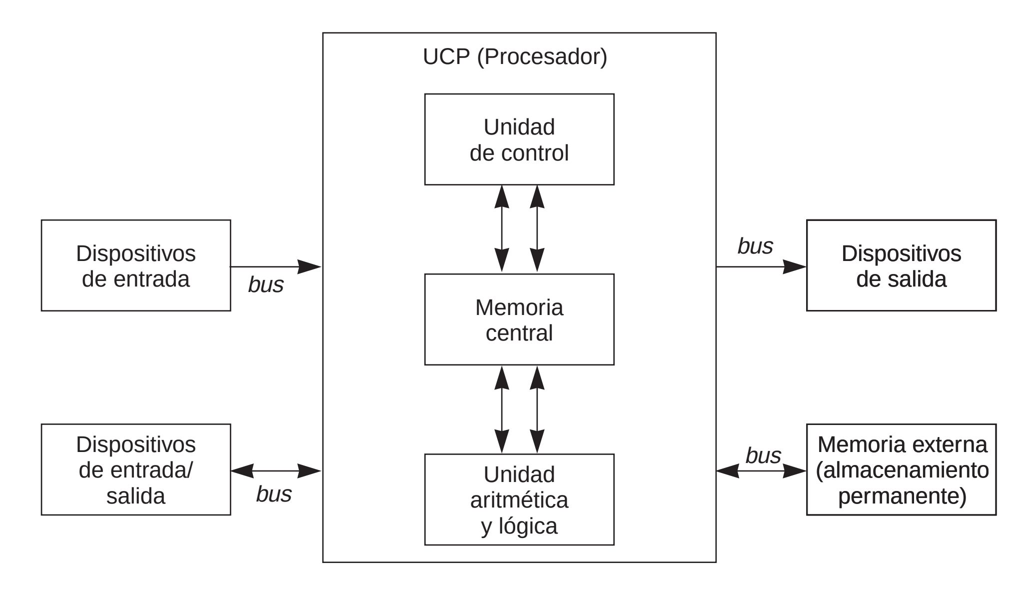


Figura 1.3. Organización física de una computadora.

1.2.1. Dispositivos de Entrada/Salida (E/S)

Los dispositivos de *Entrada/Salida* (**E/S**) (en inglés, *Input/Output* **I/O**) o *periféricos* permiten la comunicación entre la computadora y el usuario.

Los dispositivos de entrada, como su nombre indica, sirven para introducir datos (información) en la computadora para su proceso. Los datos se *leen* de los dispositivos de entrada y se almacenan en la memoria central o interna. Los dispositivos de entrada convierten la información de entrada en señales eléctricas que se almacenan en la memoria central. Dispositivos de entrada típicos son **teclados**, **lectores de tarjetas perforadas** —ya en desuso—, **lápices ópticos**, **palancas de mando** (*joystick*), **lectores de códigos de barras**, **escáneres**, **micrófonos**, **lectores de tarjetas digitales**, **lectores RFID** (tarjetas de identificación por radio frecuencia), etc. Hoy día tal vez el dispositivo de entrada más popular es el **ratón** (***mouse***) que mueve un puntero gráfico (electrónico) sobre la pantalla que facilita la interacción usuario-máquina[2].

Los *dispositivos de salida* permiten representar los resultados (salida) del proceso de los datos. El dispositivo de salida típico es la **pantalla (CRT)**[3] o **monitor**. Otros dispositivos de salida son: **impresoras** (imprimen resultados en papel), **trazadores gráficos** (*plotters*), **reconocedores** *(sintetizadores)* **de voz**, **altavoces**, etc.

Dispositivos de entrada/salida y *dispositivos de almacenamiento* masivo o auxiliar (memoria externa) son: unidad de discos (disquetes, **CD-ROM**, **DVD**, cintas, discos duros, etc.), videocámaras, memorias flash, **USB**, etc.



Figura 1.4. Dispositivo de salida (Impresora HP Color LaserJet 2600n).

1.2.2. La memoria central (interna)

La **memoria central** o simplemente **memoria** (*interna o principal*) se utiliza para almacenar información (**RAM**, **R**andom, **A**ccess **M**emory). En general, la información almacenada en memoria puede ser de dos tipos: *instrucciones*, de un programa y *datos* con los que operan las instrucciones. Por ejemplo, para que un programa se pueda *ejecutar* (correr, rodar, funcionar..., en inglés, *run*), debe ser situado en la memoria central, en una operación denominada *carga* (*load*) del programa. Después, cuando se ejecuta (se realiza, funciona) el programa, cualquier dato a procesar por el programa se debe llevar a la memoria mediante las instrucciones del programa. En la memoria central, hay también datos diversos y espacio de almacenamiento temporal que necesita el programa cuando se ejecuta a fin de poder funcionar.

---

[2] Todas las acciones a realizar por el usuario se realizarán con el ratón con la excepción de las que se requieren de la escritura de datos por teclado.

[3] *Cathode Ray Tube:* Tubo de rayos catódicos.

Ejecución

Cuando un programa se ejecuta (realiza, funciona) en una computadora, se dice que se *ejecuta*.

Con el objetivo de que el procesador pueda obtener los datos de la memoria central más rápidamente, normalmente todos los procesadores actuales (muy rápidos) utilizan una *memoria* denominada *caché* que sirve para almacenamiento intermedio de datos entre el procesador y la memoria principal. La memoria *caché* —en la actualidad— se incorpora casi siempre al procesador.

Organización de la memoria

La memoria central de una computadora es una zona de almacenamiento organizada en centenares o millares de unidades de almacenamiento individual o celdas. La memoria central consta de un conjunto de *celdas de memoria* (estas celdas o posiciones de memoria se denominan también *palabras*, aunque no "guardan" analogía con las palabras del lenguaje). El número de celdas de memoria de la memoria central, dependiendo del tipo y modelo de computadora; hoy día el número suele ser millones (512, 1.024, etc.). Cada celda de memoria consta de un cierto número de bits (normalmente 8, un *byte*).

La unidad elemental de memoria se llama *byte* (octeto). Un *byte* tiene la capacidad de almacenar un carácter de información, y está formado por un conjunto de unidades más pequeñas de almacenamiento denominadas *bits*, que son dígitos binarios (0 o 1).



Figura 1.5. Computadora portátil digital.

Por definición, se acepta que un byte contiene ocho bits. Por consiguiente, si se desea almacenar la frase:

```
Hola Mortimer todo va bien
```

la computadora utilizará exactamente 27 bytes consecutivos de memoria. Obsérvese que, además de las letras, existen cuatro espacios en blanco y un punto (un espacio es un carácter que emplea también un byte). De modo similar, el número del pasaporte

```
P57487891
```

ocupará 9 bytes, pero si se almacena como

```
P5-748-7891
```

ocupará 11. Estos datos se llaman *alfanuméricos*, y pueden constar de letras del alfabeto, dígitos o incluso caracteres especiales (símbolos: `$, #, *,` etc.).

Mientras que cada carácter de un dato alfanumérico se almacena en un byte, la información numérica se almacena de un modo diferente. Los datos numéricos ocupan 2, 4 e incluso 8 bytes consecutivos, dependiendo del tipo de dato numérico (se verá en el Capítulo 12).

Existen dos conceptos importantes asociados a cada celda o posición de memoria: su *dirección* y su *contenido*. Cada celda o byte tiene asociada una única *dirección* que indica su posición relativa en memoria y mediante la cual se puede acceder a la posición para almacenar o recuperar información. La información almacenada en una posición de memoria es su *contenido*. La Figura 1.6 muestra una memoria de computadora que consta de 1.000 posiciones en memoria con direcciones de 0 a 999. El contenido de estas direcciones o posiciones de memoria se llaman *palabras*, de modo que existen palabras de 8, 16, 32 y 64 bits. Por consiguiente, si trabaja con una máquina de 32 bits, significa que en cada posición de memoria de su computadora puede alojar 32 bits, es decir, 32 dígitos binarios, bien ceros o unos.

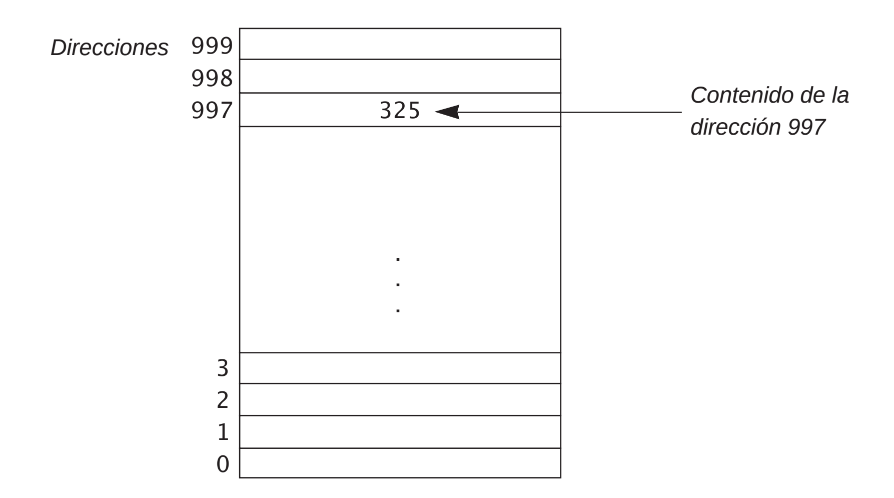


Figura 1.6. Memoria central de una computadora.

Siempre que se almacena una nueva información en una posición, se destruye (desaparece) cualquier información que en ella hubiera y no se puede recuperar. La dirección es permanente y única, el contenido puede cambiar mientras se ejecuta un programa.

La memoria central de una computadora puede tener desde unos centenares de millares de bytes hasta millones de bytes. Como el byte es una unidad elemental de almacenamiento, se utilizan múltiplos de potencia de 2 para definir el tamaño de la memoria central: *Kilo*-byte (**KB** o **Kb**) igual a 1.024 bytes (2^{10}) —prácticamente se consideran 1.000—; *Megabyte* (**MB** o **Mb**) igual a 1.024 × 1.024 bytes = = 1.048.576 (2^{20}) —prácticamente se consideran 1.000.000; *Gigabyte* (**GB** o **Gb**) igual a 1.024 MB (2^{30}), 1.073.741.824 = prácticamente se consideran 1.000 millones de MB.

Tabla 1.1. Unidades de medida de almacenamiento.

Byte	**Byte (B)**	*equivale a*	8 bits
Kilobyte	**Kbyte (KB)**	*equivale a*	1.024 bytes
Megabyte	**Mbyte (MB)**	*equivale a*	1.024 Kbytes
Gigabyte	**Gbyte (GB)**	*equivale a*	1.024 Mbytes
Terabyte	**Tbyte (TB)**	*equivale a*	1.024 Gbytes
1 **Tb** = 1.024 **Gb** = 1.024 × 1.024 **Mb** = 1.048.576 **Kb** = 1.073.741.824 **B**			

En la actualidad las computadoras personales tipo PC suelen tener memorias centrales de 512 MB a 2 GB, aunque ya es muy frecuente verlas con memorias de 4 GB y hasta 8 GB.

La memoria principal es la encargada de almacenar los programas y datos que se están ejecutando y su principal característica es que el acceso a los datos o instrucciones desde esta memoria es muy rápido.

En la memoria principal se almacenan:

- Los datos enviados para procesarse desde los dispositivos de entrada.
- Los programas que realizarán los procesos.
- Los resultados obtenidos preparados para enviarse a un dispositivo de salida.

Tipos de memoria principal

En la memoria principal se pueden distinguir dos tipos de memoria: **RAM** y **ROM**. La memoria **RAM** (**R**andom **A**ccess **M**emory, Memoria de acceso aleatorio) almacena los datos e instrucciones a procesar. Es un tipo de memoria volátil (su contenido se pierde cuando se apaga la computadora); esta memoria es, en realidad, la que se suele conocer como memoria principal o de trabajo; en esta memoria se pueden escribir datos y leer de ella. La memoria **ROM** (**R**ead **O**nly **M**emory, Memoria de sólo lectura) es una memoria permanente en la que no se puede escribir (viene pregrabada por el fabricante); es una ***memoria de sólo lectura***. Los programas almacenados en ROM no se pierden al apagar la computadora y cuando se enciende, se lee la información almacenada en esta memoria. Al ser esta memoria de sólo lectura, los programas almacenados en los chips ROM no se pueden modificar y suelen utilizarse para almacenar los programas básicos que sirven para arrancar la computadora.

1.2.3. La Unidad Central de Proceso (UCP): el Procesador

La *Unidad Central de Proceso*, **UCP** (*Central Processing Unit*, **CPU**, en inglés), dirige y controla el proceso de información realizado por la computadora. La **UCP** procesa o manipula la información almacenada en memoria; puede recuperar información desde memoria (esta información son datos o instrucciones: programas). También puede almacenar los resultados de estos procesos en memoria para su uso posterior.

La **UCP** consta de dos componentes: *unidad de control* (**UC**) y *unidad aritmética-lógica* (**UAL**) (Figura 1.7). La **unidad de control** (*Control Unit*, **CU**) coordina las actividades de la computadora y determina qué operaciones se deben realizar y en qué orden; asimismo controla y sincroniza todo el proceso de la computadora. La **unidad aritmético-lógica** (*Arithmetic-Logic Unit*, **ALU**) realiza operaciones aritméticas y lógicas, tales como suma, resta, multiplicación, división y comparaciones. Los datos en la memoria central se pueden *leer* (recuperar) o *escribir* (cambiar) por la UCP.

1.2.4. El microprocesador

El **microprocesador** es un *chip* (**un circuito integrado**) que controla y realiza las funciones y operaciones con los datos. Se suele conocer como *procesador* y es el cerebro y corazón de la computadora. En realidad el microprocesador representa a la Unidad Central de Proceso de una computadora.

El primer microprocesador comercial, el Intel 4004 fue presentado el 15 de noviembre de 1971. Existen diferentes fabricantes de microprocesadores, como Intel, Zilog, AMD, Motorola; Cyrix, etc. Microprocesadores históricos de 8 bits son el Intel 8080, Zilog Z80 o Motorola 6800; otros microprocesadores muy populares han sido: 8086, 8088 de Intel o el Motorola MC68000. En la década de los ochenta eran populares: Intel 80286, 80386, 80486; Motorola, MC 68020, MC68400; AMD 80386, 80486.

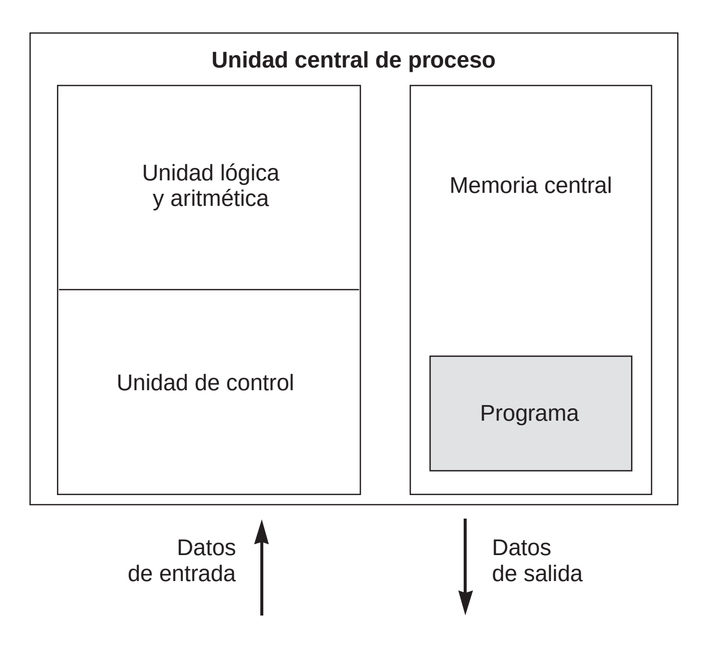


Figura 1.7. Unidad central de proceso.

En el año 1993 aparecieron el Intel Pentium y durante esa década, Intel Pentium Pro, Intel Pentium II/III y AMD K6. En 2000, Intel y **AMD** controlan el mercado con Intel Pentium IV, Intel Titanium, Intel Pentium D o bien AMD Athlon XP, AMD Duxor. En los años 2005 y 2006 aparecen las nuevas tecnologías **Intel Core Duo**, **AMD Athlon 64**, etc.

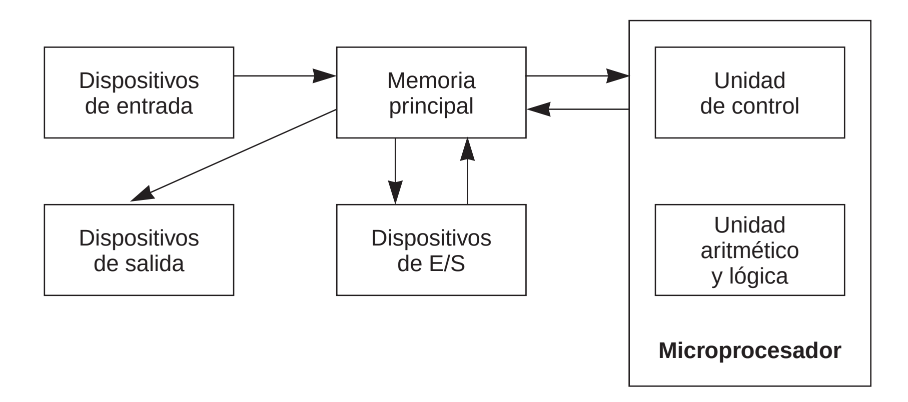


Figura 1.8. Organización física de una computadora con un microprocesador.

1.2.5. Memoria externa: almacenamiento masivo

Cuando un programa se ejecuta, se debe situar primero en memoria central de igual modo que los datos. Sin embargo, la información almacenada en la memoria se pierde (borra) cuando se *apaga* (desconecta de la red eléctrica) la computadora, y por otra parte la memoria central es limitada en capacidad. Por esta razón, para poder disponer de almacenamiento permanente, tanto para programas como para datos, se necesitan *dispositivos de almacenamiento secundario, auxiliar* o *masivo* ("*mass storage*" o "*secondary storage*").

Los **dispositivos de almacenamiento** o **memorias auxiliares** (*externas* o *secundarias*) más comúnmente utilizados son: *cintas magnéticas*, *discos magnéticos*, *discos compactos* (**CD-ROM, C**ompact **D**isk **R**ead **O**nly **M**emory) y videodiscos digitales (**DVD**). Las cintas son utilizadas principalmente por

sistemas de computadores grandes similares a las utilizadas en los equipos de audio. Los discos y disquetes magnéticos se utilizan por todas las computadoras, especialmente las medias y pequeñas —las computadoras personales. Los discos pueden ser *duros*, de gran capacidad de almacenamiento (su capacidad actual oscila entre 40 GB y 500 GB), **disquetes** o *discos flexibles* ("floppy disk"), ya casi en desuso. Aunque todavía se suelen comercializar lectoras de disquetes para compatibilidad con equipos antiguos. El disquete, ya casi en desuso, es de 3,5'' y de 1,44 MB de capacidad.



Figura 1.9. Memorias auxiliares: Tarjeta compact flash (izquierda), memoria flash USB (centro) y disco duro (derecha).

Los discos compactos (conocidos popularmente como **CD**) son soportes digitales ópticos utilizados para almacenar cualquier tipo de información (audio, vídeo, documentos...). Se desarrolló en 1980 y comenzó a comercializarse en 1982. Existen diferentes modelos CD-ROM (de sólo lectura), **CD-R** (grabable), **CD-RW** (reescribible). Su capacidad de almacenamiento va de 650 MB a 875 MB e incluso 215 MB.

Los **DVD** constituyen un formato multimedia de almacenamiento óptico y que se puede usar para guardar datos, incluyendo películas de alta calidad de vídeo y audio. Los formatos más populares son: DVD–ROM, DVD⇓R, DVD⇓RW, DVD⇓RAM, y sus capacidades de almacenamiento van desde 4,7 GB y 8,5 GB hasta 17,1 GB, según sean de una cara, de dos caras y de una capa simple o capa doble.

Los últimos discos ópticos presentados en el mercado durante 2006 son: ***Blu-ray*** y **HD DVD**. Estos discos son de alta definición y su capacidad de almacenamiento es muy grande de 15 GB a 50 GB y podrá llegar en el futuro hasta 200 GB.

La información almacenada en la memoria central es *volátil* (desaparece cuando se apaga la computadora) mientras que la información almacenada en la memoria externa (masiva) es *permanente*.

Esta información se organiza en unidades independientes llamadas **archivos** (**ficheros**, *file* en inglés). Los resultados de los programas se pueden guardar como *archivos de datos* y los programas que se escriben se guardan como *archivos de programas*, ambos en la memoria auxiliar. Cualquier tipo de archivo se puede transferir fácilmente desde la memoria auxiliar hasta la memoria central para su proceso posterior.

En el campo de las computadoras es frecuente utilizar la palabra memoria y almacenamiento o memoria externa, indistintamente. En este libro —y recomendamos su uso— se utilizará el término memoria sólo para referirse a la memoria central.

Comparación de la memoria central y la memoria externa

La memoria central o principal es mucho más rápida y cara que la memoria externa. Se deben transferir los datos desde la memoria externa hasta la memoria central, antes de que puedan ser procesados. Los datos en memoria central son: *volátiles* y desaparecen cuando se *apaga* la computadora. Los datos en memoria externa son *permanentes* y no desaparecen cuando se *apaga* la computadora.

Las computadoras modernas necesitan comunicarse con otras computadoras. Si la computadora se conecta con una *tarjeta de red* se puede conectar a una red de datos locales (*red de área local*). De este

modo se puede acceder y compartir a cada una de las memorias de disco y otros dispositivos de entrada y salida. Si la computadora tiene un *módem*, se puede comunicar con computadoras lejanas. Se pueden conectar a una red de datos o *enviar correo electrónioco* a través de las redes corporativas Intranet/Extranet o la propia red Internet. Las redes inalámbricas permiten conexiones a Internet desde númerosos lugares, siempre que su PC disponga de tarjetas o conexiones inalámbricas.

1.2.6. La computadora personal ideal para programación

Hoy día el estudiante de informática o de computación y mucho más el profesional, dispone de un amplio abanico de computadoras a precios asequibles y con prestaciones altas. En el segundo semestre de 2006 se pueden encontrar computadoras personales (PC) con 1.024 MB de memoria, grabadoras de DVD de doblecapa, procesador Pentium de 3.00/3.20 GH2 y un monitor plano 17", disco duro de 200/400 GB por 800 a 1.000 € e incluso más económicos. En computadoras pórtatíles se pueden encontrar modelos de 1024 MB, 60-80 GB, procesadores Intel Pentium de 1,736 GHz, 1,83 GHz, pantalla de 15,4" y con precios de 800 a 1200 €. La Tabla 1.2 resume nuestra propuesta y recomendación de características medias para un/una PC, a mediados de 2006.

Tabla 1.2. Características medias de una computadora portatil (*laptop*).

Procesador	Intel Centrino 1.6 a 1,73/2.0 GHz; Intel CorneDuo (1.73/1.83 GHz; AMD Turion 64).
Memoria	512 MB a 1.0 GB/2 GB.
Disco duro	60-120 GB.
Internet	Tarjeta de red; modem 56 Kbps; red inalámbrica 802,11 b/g; *Bluetooth*.
Vídeo	Memoria de vídeo de 128 a 512 MB.
Pantalla	15", 15,4" o 17" (se comercializan también de 11", 12" y 13").
Almacenamiento	Grabadora DVD +/– RW de doble capa.
Puertos	3/4 puertos USB 2.0, 1 IEEE, lector de tarjetas.
Marcas	HP, Compaq, Dell, IBM, EI System, Gateways, Acer, Toshiba, Sony, Cofiman...

1.3. REPRESENTACIÓN DE LA INFORMACIÓN EN LAS COMPUTADORAS

Una computadora es un sistema para procesar información de modo automático. Un tema vital en el proceso de funcionamiento de una computadora es estudiar la forma de representación de la información en dicha computadora. Es necesario considerar cómo se puede codificar la información en patrones de bits que sean fácilmente almacenables y procesables por los elementos internos de la computadora.

Las formas de información más significativas son: textos, sonidos, imágenes y valores numéricos y, cada una de ellas presentan peculiaridades distintas. Otros temas importantes en el campo de la programación se refieren a los métodos de detección de errores que se puedan producir en la transmisión o almacenamiento de la información y a las técnicas y mecanismos de comprensión de información al objeto de que ésta ocupe el menor espacio en los dispositivos de almacenamiento y sea más rápida su transmisión.

1.3.1. Representación de textos

La información en formato de texto se representa mediante un código en el que cada uno de los distintos símbolos del texto (tales como letras del alfabeto o signos de puntuación) se asignan a un único patrón de bits. El texto se representa como una cadena larga de bits en la cual los sucesivos patrones representan los sucesivos símbolos del texto original.

En resumen, se puede representar cualquier información escrita (texto) mediante caracteres. Los caracteres que se utilizan en computación suelen agruparse en cinco categorías:

1. **Caracteres alfabéticos** (letras mayúsculas y minúsculas, en una primera versión del abecedario inglés).

   `A, B, C, D, E, ... X, Y, Z, a, b, c, ... , X, Y, Z`

2. **Caracteres numéricos** (dígitos del sistema de numeración).

   `0, 1, 2, 3, 4, 5, 6, 7, 8, 9` *sistema decimal*

3. **Caracteres especiales** (símbolos ortográficos y matemáticos no incluidos en los grupos anteriores).

   `{ } Ñ ñ ! ? & > # ç` ...

4. **Caracteres geométricos y gráficos** (símbolos o módulos con los cuales se pueden representar cuadros, figuras geométricas, iconos, etc.

   │ ┐ ╔ → ~ ...

5. **Caracteres de control** (representan órdenes de control como el carácter para pasar a la siguiente línea [NL] o para ir al comienzo de una línea [**RC**, *retorno de carro*, «*carriage* return, **CR**»] emitir un pitido en el terminal [BEL], etc.).

Al introducir un texto en una computadora, a través de un periférico, los caracteres se codifican según un **código de entrada/salida** de modo que a cada carácter se le asocia una determinada combinación de *n* bits.

Los códigos más utilizados en la actualidad son: **EBCDIC**, **ASCII** y **Unicode**.

- **Código EBCDIC** (*Extended Binary Coded Decimal Inter Change Code*).
  Este código utiliza $n = 8$ bits de forma que se puede codificar hasta $m = 2^8 = 256$ símbolos diferentes. Éste fue el primer código utilizado para computadoras, aceptado en principio por IBM.
- **Código ASCII** (*American Standard Code for Information Interchange*).
  El código ASCII básico utiliza 7 bits y permite representar 128 caracteres (letras mayúsculas y minúsculas del alfabeto inglés, símbolos de puntuación, dígitos 0 a 9 y ciertos controles de información tales como retorno de carro, salto de línea, tabulaciones, etc.). Este código es el más utilizado en computadoras, aunque el ASCII ampliado con 8 bits permite llegar a 2^8 (256) caracteres distintos, entre ellos ya símbolos y caracteres especiales de otros idiomas como el español.
- **Código Unicode**
  Aunque ASCII ha sido y es dominante en los caracteres se leen como referencia, hoy día se requiere de la necesidad de representación de la información en muchas otras lenguas, como el portugués, español, chino, el japonés, el árabe, etc. Este código utiliza un patrón único de 16 bits para representar cada símbolo, que permite 2^{16} bits o sea hasta 65.536 patrones de bits (símbolos) diferentes.

Desde el punto de vista de unidad de almacenamiento de caracteres, se utiliza el archivo (**fichero**). Un **archivo** consta de una secuencia de símbolos de una determinada longitud codificados utilizando ASCII o Unicode y que se denomina **archivo de texto**. Es importante diferenciar entre archivos de texto simples que son manipulados por los programas de utilidad denominados **editores de texto** y los archivos de texto más elaborados que se producen por los procesadores de texto, tipo Microsoft Word. Ambos constan de caracteres de texto, pero mientras el obtenido con el editor de texto, es un archivo de texto puro que codifica carácter a carácter, el archivo de texto producido por un procesador de textos contiene números, códigos que representan cambios de formato, de tipos de fuentes de letra y otros, e incluso pueden utilizar códigos propietarios distintos de ASCII o Unicode.

1.3.2. Representación de valores númericos

El almacenamiento de información como caracteres codificados es ineficiente cuando la información se registra como numérica pura. Veamos esta situación con la codificación del número 65; si se almacena como caracteres ASCII utilizando un byte por símbolo, se necesita un total de 16 bits, de modo que el número mayor que se podía almacenar en 16 bits (dos bytes) sería 99. Sin embargo, si utilizamos *notación binaria* para almacenar enteros, el rango puede ir de 0 a 65.535 (2^{16} – 1) para números de 16 bits. Por consiguiente, la notación binaria (o variantes de ellas) es la más utilizada para el almacenamiento de datos numéricos codificados.

La solución que se adopta para la representación de datos numéricos es la siguiente: al introducir un número en la computadora se codifica y se almacena como un texto o cadena de caracteres, pero dentro del programa a cada dato se le envía un tipo de dato específico y es tarea del programador asociar cada dato al tipo adecuado correspondiente a las tareas y operaciones que se vayan a realizar con dicho dato.

El método práctico realizado por la computadora es que una vez definidos los datos numéricos de un programa, una rutina (función interna) de la biblioteca del compilador (traductor) del lenguaje de programación se encarga de transformar la cadena de caracteres que representa el número en su notación binaria.

Existen dos formas de representar los datos numéricos: números enteros o números reales.

Representación de enteros

Los datos de tipo entero se representan en el interior de la computadora en notación binaria. La memoria ocupada por los tipos enteros depende del sistema, pero normalmente son dos, bytes (en las versiones de MS-DOS y versiones antiguas de Windows y cuatro bytes en los sistemas de 32 bits como Windows o Linux). Por ejemplo, un entero almacenado en 2 bytes (16 bits):

```
1000 1110 0101 1011
```

Los enteros se pueden representar con signo (`signed`, en C++) o sin signo (`unsigned`, en C++); es decir, números positivos o negativos. Normalmente, se utiliza un bit para el signo. Los enteros sin signo al no tener signo pueden contener valores positivos más grandes. Normalmente, si un entero no se especifica «con/sin signo» se suele asignar con signo por defecto u omisión.

El rango de posibles valores de enteros depende del tamaño en bytes ocupado por los números y si se representan con signo o sin signo (la Tabla 1.3 resume características de tipos estándar en C++).

Representación de reales

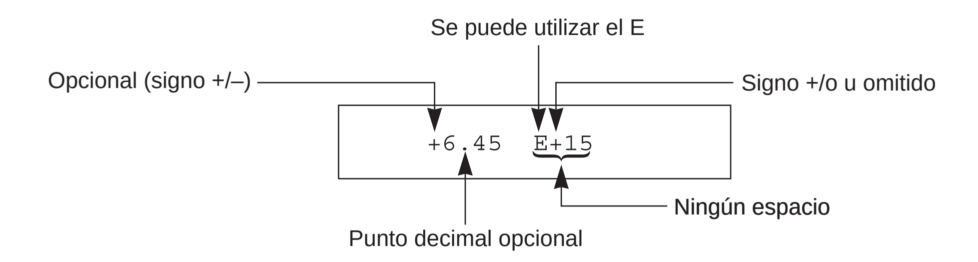
Los números reales son aquellos que contienen una parte decimal como `2,6` y `3,14152`. Los reales se representan en *notación científica* o en *coma flotante*; por esta razón en los lenguajes de programación, como C++, se conocen como números en coma flotante.

Existen dos formas de representar los números reales. La primera se utiliza con la notación del punto decimal (*ojo* en el formato de representación español de números decimales, la parte decimal se representa por coma).

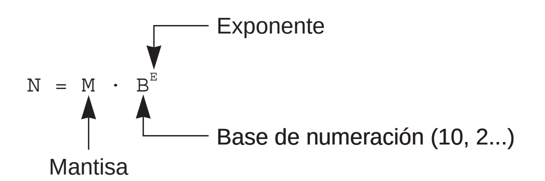
Ejemplos

```
12.35  99901.32  0.00025  9.0
```

La segunda forma para representar números en coma flotante en la notación científica o exponencial, conocida también como notación *E*. Esta notación es muy útil para representar números muy grandes o muy pequeños.



Notación exponencial



Ejemplos

2.52 e + 8		
8.34 E – 4	*equivale a*	$8.34/10^4$ = 0.000834
7E5		
–18.35e15		
5.95E25		
9.11e – 31	*equivale a*	0.000000000000000000000000000000911

Representación de caracteres

Un documento de texto se escribe utilizando un conjunto de caracteres adecuado al tipo de documento. En los lenguajes de programación se utilizan, principalmente, dos códigos de caracteres. El más común es **ASCII** (American Standard Code for Information Interchange) y algunos lenguajes, tal como Java, utilizan **Unicode** (*www.unicode.org*). Ambos códigos se basan en la asignación de un código numérico a cada uno de los tipos de caracteres del código.

En C++, los *caracteres* se procesan normalmente usando el tipo `char`, que asocia cada carácter a un código numérico que se almacena en un byte.

El código ASCII básico que utiliza 7 bits (128 caracteres distintos) y el ASCII *ampliado* a 8 bits (256 caracteres distintos) son los códigos más utilizados. Así se pueden representar caracteres tales como `'A'`, `'B'`, `'c'`, `'$'`, `'4'`, `'5'`, etc.

En el caso de C++ se considera un dato de tipo carácter denominado *wchar_t* que sirve para representar caracteres de tipo internacional para manejar lenguajes extranjeros.

La Tabla 1.3, recoge los tipos enteros, reales y carácter utilizados en C++, la memoria utilizada (número de bytes ocupados por el dato) y el rango de números.

1.3.3. Representación de imágenes

Las imágenes se adquieren mediante periféricos especializados tales como escáneres, cámaras digitales de vídeo, cámaras fotográficas, etc. Una imagen, al igual que otros tipos de información, se representan por patrones de bits, generados por el periférico correspondiente. Existen dos métodos básicos para representar imágenes: *mapas de bits* y *mapas de vectores*.

Tabla 1.3. Tipos enteros reales, en C++.

		Carácter y bool
Tipo	**Tamaño**	**Rango**
`short (short int)`	2 bytes	`–32.738..32.767`
`int`	4 bytes	`–2.147.483.648` a `2.147.483.647`
`long (long int)`	4 bytes	`–2.147.483.648` a `2.147.483.647`
`float (real)`	4 bytes	10^{-38} a 10^{38} (*aproximadamente*)
`double`	8 bytes	10^{-308} a 10^{308} (*aproximadamente*)
`long double`	10 bytes	10^{-4932} a 10^{4932} (*aproximadamente*)
`char (carácter)`	1 byte	Todos los caracteres ASCII
`bool`	1 byte	`True` (verdadero) y `false` (falso)

En las técnicas de *mapas de bits*, una imagen se considera como una colección de puntos, cada uno de los cuales se llama *pixel* (abreviatura de «*picture element*»). Una imagen en blanco y negro se representa como una cadena larga de bits que representan las filas de píxeles en la imagen, donde cada bit es bien `1` o bien `0`, dependiendo de que el pixel correspondiente sea blanco o negro. En el caso de imágenes en color, cada pixel se representa por una combinación de bits que indican el color de los pixel. Cuando se utilizan técnicas de mapas de bits, el patrón de bits resultante se llama *mapa de bits*, significando que el patrón de bits resultante que representa la imagen es poco más que un mapa de la imagen.

Muchos de los periféricos de computadora —tales como cámaras de vídeo, escáneres, etc.— convierten imágenes de color en formato de mapa de bits. Los formatos más utilizados en la representación de imágenes se muestran en la Tabla 1.4.

Tabla 1.4. Mapas de bits.

Formato	**Origen y descripción**
`BMP`	**Microsoft**. Formato sencillo con imágenes de gran calidad pero con el inconveniente de ocupar mucho (no útil para la web).
`JPEG`	Grupo **JPEG**. Calidad aceptable para imágenes naturales. Incluye compresión. Se utiliza en la web.
`GIF`	**CompuServe**. Muy adecuado para imágenes no naturales (logotipos, banderas, dibujos anidados...). Muy usuado en la web.

Mapas de vectores. Otros métodos de representar una imagen se fundamentan en descomponer la imagen en una colección de objetos tales como líneas, polígonos y textos con sus respectivos atributos o detalles (grosor, color, etc.).

Tabla 1.5. Mapas de vectores.

Formato	**Descripción**
`IGES`	ASME/ANSI. Estándar para intercambio de datos y modelos de (AutoCAD,...).
`Pict`	Apple Computer. Imágenes vectoriales.
`EPS`	Adobe Computer.
`TrueType`	Apple y Microsoft para EPS.

1.3.4. Rrepresentación de sonidos

La representación de sonidos ha adquirido una importancia notable debido esencialmente a la infinidad de aplicaciones multimedia tanto autónomas como en la *web*.

El método más genérico de codificación de la información de audio para almacenamiento y manipulación en computadora es mostrar la amplitud de la onda de sonido en intervalos regulares y registrar las series de valores obtenidos. La señal de sonido se capta mediante micrófonos o dispositivos similares y produce una señal analógica que puede tomar cualquier valor dentro de un intervalo continuo determinado. En un intervalo de tiempo continuo se dispone de infinitos valores de la señal analógica, que es necesario almacenar y procesar, para lo cual se recurre a una *técnica de muestreo*. Las muestras obtenidas se digitalizan con un conversor analógico-digital, de modo que la señal de sonido se representa por secuencias de bits (por ejemplo, 8 o 16) para cada muestra. Esta técnica es similar a la utilizada, históricamente, por las comunicaciones telefónicas a larga distancia. Naturalmente, dependiendo de la calidad de sonido que se requiera, se necesitarán más números de bits por muestra, frecuencias de muestreo más altas y lógicamente más muestreos por períodos de tiempo[4].

Como datos de referencia puede considerar que para obtener reproducción de calidad de sonido de alta fidelidad para un disco CD de música, se suele utilizar, al menos, una frecuencia de muestreo de 44.000 muestras por segundo. Los datos obtenidos en cada muestra se codifican en 16 bits (32 bits para grabaciones en estéreo). Como dato anecdótico, cada segundo de música grabada en estéreo requiere más de un millón de bits.

Un sistema de codificación de música muy extendido en sintetizadores musicales es MIDI (***M****usical* ***I****nstruments* ***D****igital* ***I****nterface*) que se encuentra en sintetizadores de música para sonidos de videojuegos, sitios web, teclados electrónicos, etc.

1.4. CONCEPTO DE ALGORITMO

El objetivo fundamental de este texto es enseñar a resolver problemas mediante una computadora. El programador de computadora es antes que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A lo largo de todo el libro nos referiremos a la *metodología necesaria para resolver problemas mediante programas*, concepto que se denomina **metodología de la programación**. El eje central de esta metodología es el concepto, ya tratado, de algoritmo.

Un algoritmo es un método para resolver un problema. Aunque la popularización del término ha llegado con el advenimiento de la era informática, **algoritmo** proviene de *Mohammed al-Khowârizmi*, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra *algorismus* derivó posteriormente en algoritmo. Euclides, el gran matemático griego (del siglo IV antes de Cristo) que inventó un método para encontrar el máximo común divisor de dos números, se considera con Al-Khowârizmi el otro gran padre de la algoritmia (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth —inventor de Pascal, Modula-2 y Oberon— tituló uno de sus más famosos libros, *Algoritmos + Estructuras de datos = Programas*, significándonos que sólo se puede llegar a realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será una de las hipótesis fundamentales consideradas en esta obra.

La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.

---

[4] En las obras del profesor Alberto Prieto, Schaum «*Conceptos de Informática* e *Introducción a la Informática*», publicadas en McGraw-Hill, puede encontrar una excelente referencia sobre estos conceptos y otros complementarios de este capítulo introductorio.

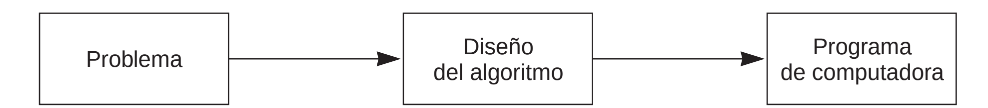


Figura 1.10. Resolución de un problema.

Los pasos para la resolución de un problema son:

1. *Diseño del algoritmo,* que describe la secuencia ordenada de pasos —sin ambigüedades— que conducen a la solución de un problema dado. (*Análisis del problema y desarrollo del algoritmo*.)
2. Expresar el algoritmo como un *programa* en un lenguaje de programación adecuado. (*Fase de codificación*.)
3. *Ejecución y validación* del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será *el diseño de algoritmos*. A la enseñanza y práctica de esta tarea denominada *algoritmia* se dedica gran parte de este libro.

El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, *la solución de un problema se puede expresar mediante un algoritmo*.

1.4.1. Características de los algoritmos

Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser *preciso* e indicar el orden de realización de cada paso.
- Un algoritmo debe estar *definido*. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada, Proceso* y *Salida*. En el algoritmo de receta de cocina citado anteriormente se tendrá:

Entrada: ingredientes y utensilios empleados.
Proceso: elaboración de la receta en la cocina.
Salida: terminación del plato (por ejemplo, cordero).

Ejemplo 1.1

Un cliente ejecuta un pedido a una fábrica. La fábrica examina en su banco de datos la ficha del cliente; si el cliente es solvente entonces la empresa acepta el pedido; en caso contrario, rechazará el pedido. Redactar el algoritmo correspondiente.

Los pasos del algoritmo son:

```
1. Inicio.
2. Leer el pedido.
3. Examinar la ficha del cliente.
4. Si el cliente es solvente, aceptar pedido; en caso contrario, rechazar
   pedido.
5. Fin.
```

Ejemplo 1.2

Se desea diseñar un algoritmo para saber si un número es primo o no.

Un número es primo si sólo puede dividirse por sí mismo y por la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo, 9, 8, 6, 4, 12, 16, 20, etc., no son primos, ya que son divisibles por números distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etc.

El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4, etc.

```
1. Inicio.
2. Poner X igual a 2 (x = 2, x variable que representa a los divisores
   del número que se busca N).
3. Dividir N por X (N/X).
4. Si el resultado de N/X es entero, entonces N es un número primo y bi-
   furcar al punto 7; en caso contrario, continuar el proceso.
5. Suma 1 a X (X ↔ X + 1).
6. Si X es igual a N, entonces N es un número primo; en caso contrario,
   bifurcar al punto 3.
7. Fin.
```

Por ejemplo, si `N` es `131`, los pasos anteriores serían:

```
1. Inicio.
2. X = 2.
3 y 4. 131/X. Como el resultado no es entero, se continúa el proceso.
5. X ↔ 2 + 1, luego X = 3.
6. Como X no es 131, se bifurca al punto 3.
3 y 4. 131/X resultado no es entero.
5. X ↔ 3 + 1, X = 4.
6. Como X no es 131 bifurca al punto 3.
3 y 4.131/X..., etc.
7. Fin.
```

Ejemplo 1.3

Realizar la suma de todos los números pares entre 2 y 1.000.

El problema consiste en sumar 2 + 4 + 6 + 8 ... + 1.000. Utilizaremos las palabras `SUMA` y `NUMERO` (*variables*, serán denominadas más tarde) para representar las sumas sucesivas (2+4), (2+4+6), (2+4+6+8), etcétera. La solución se puede escribir con el siguiente algoritmo:

```
1. Inicio.
2. establecer SUMA a 0.
3. establecer NUMERO a 2.
4. Sumar NUMERO a SUMA. El resultado será el nuevo valor de la suma
   (SUMA).
5. Incrementar NUMERO en 2 unidades.
6. Si NUMERO =< 1.000 bifurcar al paso 4;
7. en caso contrario, escribir el último valor de SUMA y terminar el pro-
   ceso.
8. Fin.
```

1.5. PROGRAMACIÓN ESTRUCTURADA

La programación orientada a objetos se desarrolló para tratar de paliar diversas limitaciones que se encontraban en anteriores enfoques de programación. Para apreciar las ventajas de la POO, es preciso constatar las limitaciones citadas y cómo se producen con los lenguajes de programación tradicionales.

C, Pascal y FORTRAN, y lenguajes similares, se conocen como *lenguajes procedimentales* (por procedimientos). Es decir, cada sentencia o instrucción señala al compilador para que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir por cinco, etc. En resumen, un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. En el caso de pequeños programas, estos principios de organización (denominados *paradigma)* se demuestran eficientes. El programador sólo tiene que crear esta lista de instrucciones en un lenguaje de programación, compilar en la computadora y ésta, a su vez, ejecuta estas instrucciones.

Cuando los programas se vuelven más grandes, cosa que lógicamente sucede cuando aumenta la complejidad del problema a resolver, la lista de instrucciones aumenta considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Los programadores pueden controlar, de modo normal, unos centenares de líneas de instrucciones. Para resolver este problema los programas se descompusieron en unidades más pequeñas que adoptaron el nombre de *funciones* (*procedimientos*, *subprogramas* o *subrutinas* en otros lenguajes de programación). De este modo en un programa orientado a procedimientos se divide en funciones, de modo que cada función tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de romper en programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas *módulos* (normalmente, en el caso de C, denominadas **archivos** o **ficheros**); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). Esta característica hace que a medida que los programas se hacen más grandes y complejos, el paradigma estructurado comienza a dar señales de debilidad y resultando muy difícil terminar los programas de un modo eficiente. Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos. Tal vez las dos razones más evidentes son éstas. Primero, las funciones tienen acceso ilimitado a los datos globales. Segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental proporcionan un modelo pobre del mundo real.

1.5.1. Datos locales y datos globales

En un programa procedimental, por ejemplo escrito en C, existen dos tipos de datos. *Datos locales* que son ocultos en el interior de la función y son utilizados, exclusivamente, por la función. Estos datos locales están estrechamente relacionados con sus funciones y están protegidos de modificaciones por otras funciones.

Otro tipo de datos son los *datos globales* a los cuales se puede acceder desde *cualquier* función del programa. Es decir, dos o más funciones pueden acceder a los mismos datos siempre que estos datos sean globales. En la Figura 1.11 se muestra la disposición de variables locales y globales en un programa procedimental.

Un programa grande (Figura 1.12) se compone de numerosas funciones y datos globales y ello conlleva una multitud de conexiones entre funciones y datos que dificulta su comprensión y lectura.

Todas estas conexiones múltiples originan diferentes problemas. En primer lugar, hacen difícil conceptuar la estructura del programa. En segundo lugar, el programa es difícil de modificar ya que cambios en datos globales pueden necesitar la reescritura de todas las funciones que acceden a los mismos. También puede suceder que estas modificaciones de los datos globales pueden no ser aceptadas por todas o algunas de las funciones.

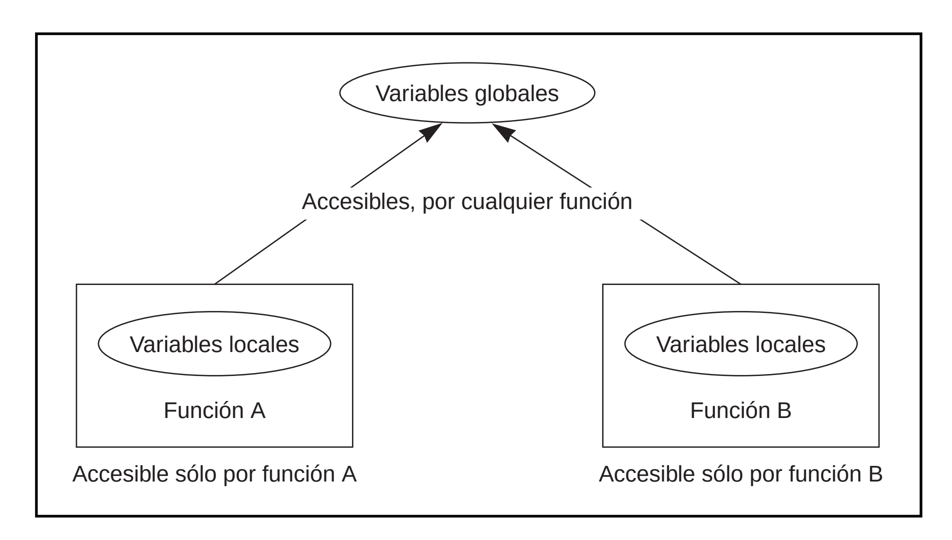


Figura 1.11. Datos locales y globales.

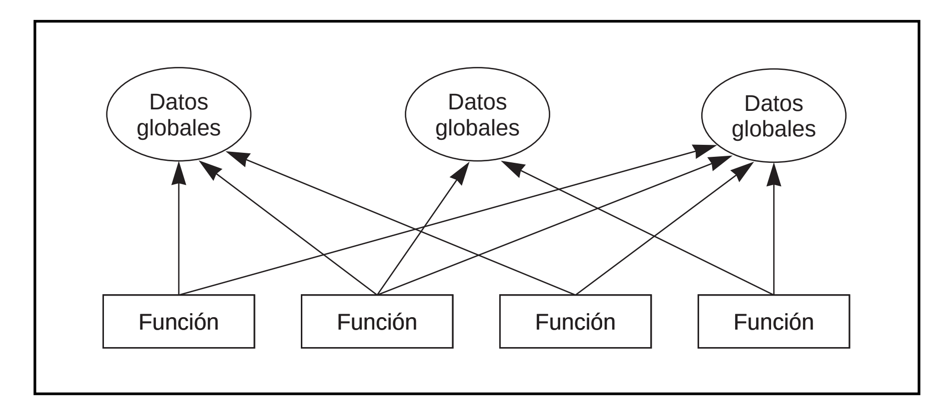


Figura 1.12. Un programa procedimental.

1.5.2 Modelado del mundo real

Un segundo problema importante de la programación estructurada reside en el hecho de que la disposición separada de datos y funciones no se corresponden con los modelos de las cosas del mundo real. En el mundo físico se trata con objetos físicos tales como personas, autos o aviones. Estos objetos no son como los datos ni como las funciones. Los objetos complejos o no del mundo real tienen *atributos* y *comportamiento.*

Los **atributos** o características de los objetos son, por ejemplo: en las personas, su edad, su profesión, su domicilio, etc.; en un auto, la potencia, el número de matrícula, el precio, número de puertas, etc; en una casa, la superficie, el precio, el año de construcción, la dirección, etc. En realidad, los atributos del mundo real tienen su equivalente en los datos de un programa; tienen un valor específico, tal como 200 metros cuadrados, 20.000 dólares, cinco puertas, etc.

El **comportamiento** es una acción que ejecutan los objetos del mundo real como respuesta a un determinado estímulo. Si usted pisa los frenos en un auto, el coche (carro) se detiene; si acelera, el auto aumenta su velocidad, etc. El comportamiento, en esencia, es como una función: se llama a una función para hacer algo (visualizar la nómina de los empleados de una empresa).

Por estas razones, ni los datos ni las funciones, por sí mismas, modelan los objetos del mundo real de un modo eficiente.

> La programación estructurada mejora la claridad, fiabilidad y facilidad de mantenimiento de los programas; sin embargo, para programas grandes o a gran escala, presentan retos de difícil solución.

1.6. PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos, tal vez el paradigma de programación más utilizado en el mundo del desarrollo de software y de la ingeniería de software del siglo XXI, trae un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación *procedimental* que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque *procedimental* de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema.

La idea fundamental de los lenguajes orientados a objetos es combinar en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama un **objeto**.

Las funciones de un objeto se llaman *funciones miembro* en C++ o *métodos* (éste es el caso de Smalltalk, uno de los primeros lenguajes orientados a objetos), y son el único medio para acceder a sus datos. Los datos de un objeto, se conocen también como *atributos* o *variables de instancia*. Si se desea leer datos de un objeto, se llama a una función miembro del objeto. Se accede a los datos y se devuelve un valor. No se puede acceder a los datos directamente. Los datos son ocultos, de modo que están protegidos de alteraciones accidentales. Los datos y las funciones se dice que están *encapsulados en una única entidad. El encapsulamiento de datos y la ocultación* de los datos son términos clave en la descripción de lenguajes orientados a objetos.

Si se desea modificar los datos de un objeto, se conoce exactamente cuáles son las funciones que interactúan con las funciones miembro del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa. Un programa C++ se compone normalmente de un número de objetos que se comunican unos con otros mediante la llamada a otras funciones miembro. La organización de un programa en C++ se muestra en la Figura 1.13. La llamada a una función miembro de un objeto se denomina *enviar un mensaje* a otro objeto.

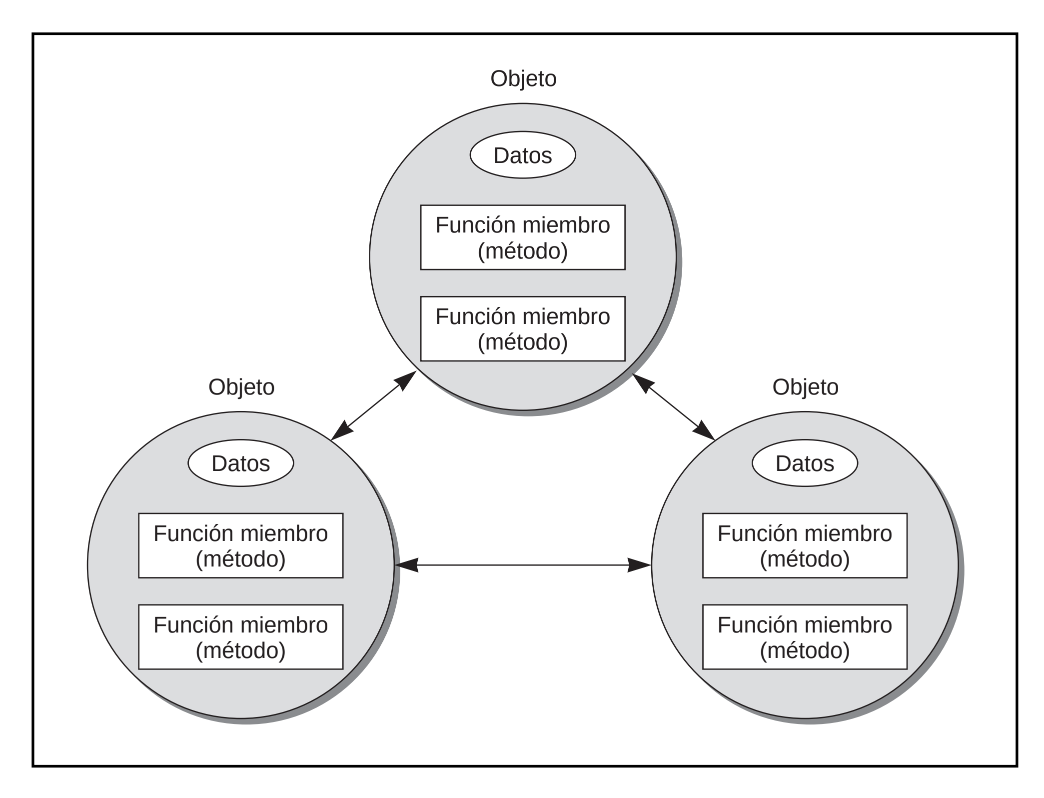


Figura 1.13. Organización típica de un programa orientado a objetos

> En el paradigma orientado a objetos, el programa se organiza como un conjunto finito de objetos que contiene datos y operaciones (funciones miembro en C++) que llaman a esos datos y que se comunican entre sí mediante mensajes.

1.6.1. Propiedades fundamentales de la orientación a objetos

Existen diversas características ligadas a la orientación a objetos. Todas las propiedades que se suelen considerar, no son exclusivas de este paradigma, ya que pueden existir en otros paradigmas, pero en su conjunto definen claramente los lenguajes orientados a objetos. Estas propiedades son:

- **Abstracción (tipos abstractos de datos y clases)**.
- **Encapsulado de datos**.
- **Ocultación de datos**.
- **Herencia**.
- **Polimorfismo**.

C++ soporta todas las características anteriores que definen la orientación a objetos, aunque hay numerosas discusiones en torno a la consideración de C++ como lenguaje orientado a objetos. La razón es que en contraste con lenguajes tales como Smalltalk, Java o C#, C++ no es un lenguaje orientado a objetos puro. C++ soporta orientación a objetos pero es compatible con C y permite que programas C++ se escriban sin utilizar características orientadas a objetos. De hecho, C++ es un lenguaje *multiparadigma* que permite programación estructurada, *procedimental*, orientada a objetos y genérica.

1.6.2. Abstracción

La abstracción es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta los restantes aspectos. El término **abstracción** que se suele utilizar en programación se refiere al hecho de diferenciar entre las propiedades externas de una entidad y los detalles de la composición interna de dicha entidad. Es la abstracción la que permite ignorar los detalles internos de un dispositivo complejo tal como una computadora, un automóvil, una lavadora o un horno de microondas, etc., y usarlo como una única unidad comprensible. Mediante la abstracción se diseñan y fabrican estos sistemas complejos en primer lugar y, posteriormente, los componentes más pequeños de los cuales están compuestos. Cada componente representa un nivel de abstracción en el cual el uso del componente se aísla de los detalles de la composición interna del componente. La abstracción posee diversos grados denominados niveles de abstracción.

En consecuencia, la abstracción posee diversos grados de complejidad que se denominan *niveles de abstracción* que ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. En el modelado orientado a objetos de un sistema esto significa centrarse en *qué* es *y qué hace* un objeto y no en *cómo* debe implementarse. Durante el proceso de abstracción es cuando se decide qué características y comportamiento debe tener el modelo.

Aplicando la abstracción se es capaz de construir, analizar y gestionar sistemas de computadoras complejos y grandes que no se podrían diseñar si se tratara de modelar a un nivel detallado. En cada nivel de abstracción se visualiza el sistema en términos de componentes, denominados **herramientas abstractas**, cuya composición interna se ignora. Esto nos permite concentrarnos en cómo cada componente interactúa con otros componentes y centrarnos en la parte del sistema que es más relevante para la tarea a realizar en lugar de perderse a nivel de detalles menos significativos.

En estructuras o registros, las propiedades individuales de los objetos se pueden almacenar en los miembros. Para los objetos es de interés *cómo* están organizados sino también *qué* se puede hacer con ellos. Es decir, las operaciones que forman la internan de un objeto son también importantes. El primer concepto en el mundo de la orientación a objetos nació con los tipos abstractos de datos (TAD). Un tipo abstracto de datos describe no sólo los atributos de un objeto, sino también su comportamiento (las operaciones). Esto puede incluir también una descripción de los estados que puede alcanzar un objeto.

Un medio de reducir la complejidad es la abstracción. Las características y los procesos se reducen a las propiedades esenciales, son resumidas o combinadas entre sí. De este modo, las características complejas se hacen más manejables.

Ejemplo 1.4

Diferentes modelos de abstracción del término coche (carro).

- Un `coche (carro)` es la combinación (o composición) de diferentes partes, tales como motor, carrocería, cuatro ruedas, cinco puertas, etc.
- Un `coche (carro)` es un concepto común para diferentes tipos de coches. Pueden clasificarse por el nombre del fabricante (Audi, BMW, SEAT, Toyota, Chrisler...), por su categoría (turismo, deportivo, todoterreno...), por el carburante que utilizan (gasolina, gasoil, gas, híbrido…).

La abstracción `coche` se utilizará siempre que la marca, la categoría o el carburante no sean significativos. Así, un `carro (coche)` se utilizará para transportar personas o ir de Carchelejo a Cazorla.

1.6.3. Encapsulación y ocultación de datos

El *encapsulado* o *encapsulación de datos* es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de los objetos que poseen las mismas características y

comportamiento se agrupan en clases, que no son más que unidades o módulos de programación que encapsulan datos y operaciones.

La ocultación de datos permite separar el aspecto de un componente, definido por su *interfaz* con el exterior, de sus detalles internos de implementación. Los términos ocultación de la información (*information hiding*) y encapsulación de datos (*data encapsulation*) se suelen utilizar como sinónimos, pero no siempre es así, y muy al contrario, son términos similares pero distintos. En C++ no es lo mismo, ya los datos internos están protegidos del exterior y no se pueden acceder a ellos más que desde su propio interior y por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.

El diseño de un programa orientado a objetos contiene, al menos, los siguientes pasos;

1. Identificar los *objetos* del sistema.
2. Agrupar en *clases* a todos objetos que tengan características y comportamiento comunes.
3. Identificar los *datos* y *operaciones* de cada una de las clases.
4. Identificar las *relaciones* que pueden existir entre las clases.

En C++, un **objeto** es un elemento individual con su propia identidad; por ejemplo, un libro, un automóvil... Una **clase** puede describir las propiedades genéricas de un ejecutivo de una empresa (nombre, título, salario, cargo...) mientras que un objeto representará a un ejecutivo específico (Luis Mackoy, director general). En general, una clase define qué datos se utilizan para representar un objeto y las operaciones que se pueden ejecutar sobre esos datos.

Cada clase tiene sus propias características y comportamiento; en general, una clase define los datos que se utilizan y las operaciones que se pueden ejecutar sobre esos datos. Una clase describe un objeto. En el sentido estricto de programación, una clase es un tipo de datos. Diferentes variables se pueden crear de este tipo. En programación orientada a objetos, éstas se llaman *instancias*. Las instancias son, por consiguiente, la realización de los objetos descritos en una clase. Estas instancias constan de datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas dentro de ellas.

Los términos *objeto* e *instancia* se utilizan frecuentemente como sinónimos (especialmente en C++). Si una variable de tipo `Carro` se declara, se crea un objeto `Carro` (una instancia de la clase `Carro`).

Las operaciones definidas en los objetos se llaman *métodos*. Cada operación llamada por un objeto se interpreta como un *mensaje* al objeto, que utiliza un método específico para procesar la operación.

En el diseño de programas orientados a objetos se realiza en primer lugar el diseño de las clases que representan con precisión aquellas cosas que trata el programa. Por ejemplo, un programa de dibujo, puede definir clases que representan rectángulos, líneas, pinceles, colores, etc. Las definiciones de clases, incluyen una descripción de operaciones permisibles para cada clase, tales como desplazamiento de un círculo o rotación de una línea. A continuación se prosigue el diseño de un programa utilizando objetos de las clases.

El diseño de clases fiables y útiles puede ser una tarea difícil. Afortunadamente, los lenguajes POO facilitan la tarea ya que incorporan clases existentes en su propia programación. Los fabricantes de software proporcionan numerosas bibliotecas de clases, incluyendo bibliotecas de clases diseñadas para simplificar la creación de programas para entornos tales como Windows, Linux, Macintosh o Unix. Uno de los beneficios reales de C++ es que permite la reutilización y adaptación de códigos existentes y ya bien probados y depurados.

1.6.4. Objetos

El objeto es el centro de la programación orientada a objetos. Un objeto es algo que se visualiza, se utiliza y juega un rol o papel. Si se programa con enfoque orientado a objetos, se intentan descubrir e implementar los objetos que juegan un rol en el dominio del problema y en consecuencia programa. La estructura interna y el comportamiento de un objetivo, en una primera fase, no tiene prioridad. Es importante que un objeto tal como un carro o una casa juegan un rol.

Dependiendo del problema, diferentes aspectos de un aspecto son relevantes. Un carro puede ser ensamblado de partes tales como un motor, una carrocería, unas puertas o puede ser descrito utilizando propiedades tales como su velocidad, su kilometraje o su fabricante. Estos atributos indican el objeto. De modo similar una persona, también se puede ver como un objeto, del cual se disponen de diferentes atributos. Dependiendo de la definición del problema, esos atributos pueden ser el nombre, apellido, dirección, número de teléfono, color del cabello, altura, peso, profesión, etc.

Un objeto no necesariamente ha de realizar algo concreto o tangible. Puede ser totalmente abstracto y también puede describir un proceso. Por ejemplo, un partido de baloncesto o de rugby puede ser descrito como un objeto. Los atributos de este objeto pueden ser los jugadores, el entrenador, la puntuación y el tiempo transcurrido de partido.

Cuando se trata de resolver un problema con orientación a objetos, dicho problema no se descompone en funciones como en programación estructurada tradicional, caso de C, sino en objetos. El pensar en términos de objetos tiene una gran ventaja: se asocian los objetos del problema a los objetos del mundo real.

¿Qué tipos de cosas son objetos en los programas orientados a objetos? La respuesta está limitada por su imaginación aunque se pueden agrupar en categorías típicas que facilitarán su búsqueda en la definición del problema de un modo más rápido y sencillo.

- RecursosHumanos:
  - Empleados.
  - Estudiantes.
  - Clientes.
  - Vendedores.
  - Socios.
- Colecciones de datos:
  - Arrays (arreglos).
  - Listas.
  - Pilas.
  - Árboles.
  - Árboles binarios.
  - Grafos.
- Tipos de datos definidos por usuarios:
  - Hora.
  - Números complejos.
  - Puntos del plano.
  - Puntos del espacio.
  - Ángulos.
  - Lados.
- Elementos de computadoras:
  - Menús.
  - Ventanas.
  - Objetos gráficos (rectángulos, círculos, rectas, puntos,…).
  - Ratón (mouse).
  - Teclado.
  - Impresora.
  - USB.
  - Tarjetas de memoria de cámaras fotográficas.
- Objetos físicos:
  - Carros.
  - Aviones.

— Trenes.
— Barcos.
— Motocicletas.
— Casas.

- Componentes de videojuegos:

— Consola.
— Mandos.
— Volante.
— Conectores.
— Memoria.
— Acceso a Internet.

La correspondencia entre objetos de programación y objetos del mundo real es el resultado eficiente de combinar datos y funciones que manipulan esos datos. Los objetos resultantes ofrecen una mejor solución al diseño del programa que en el caso de los lenguajes orientados a procedimientos.

Un **objeto** se puede definir desde el punto de vista conceptual como una entidad individual de un sistema y que se caracteriza por un estado y un comportamiento. Desde el punto de vista de implementación un **objeto** es una entidad que posee un conjunto de *datos* y un conjunto de *operaciones* (*funciones* o *métodos*).

El estado de un objeto viene determinado por los valores que toman sus datos, cuyos valores pueden tener las restricciones impuestas en la definición del problema. Los datos se denominan también *atributos* y componen la estructura del objeto y las operaciones —también llamadas *métodos*— representan los servicios que proporciona el objeto.

La representación gráfica de un objeto en **UML** se muestra en la Figura 1.14.

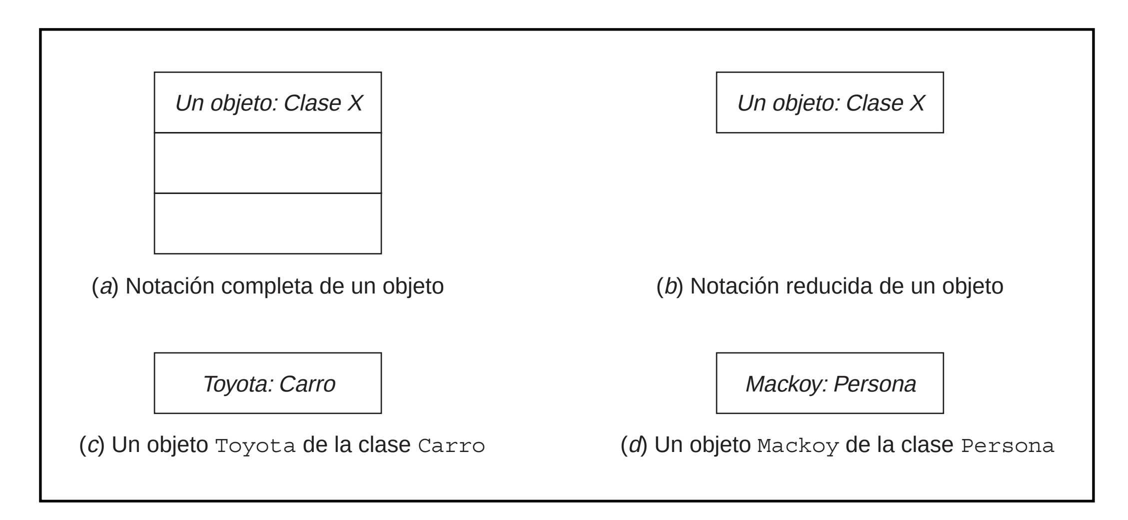


Figura 1.14. Representación de objetos en UML (Lenguaje Unificado de Modelado).

1.6.5. Clases

En POO los objetos son miembros de **clases**. En esencia, una clase es un tipo de datos al igual que cualquier otro tipo de dato definido en un lenguaje de programación. La diferencia reside en que la clase es un tipo de dato que contiene datos y funciones. Una clase contiene muchos objetos y es preciso definirla, aunque su definición no implica creación de objetos.

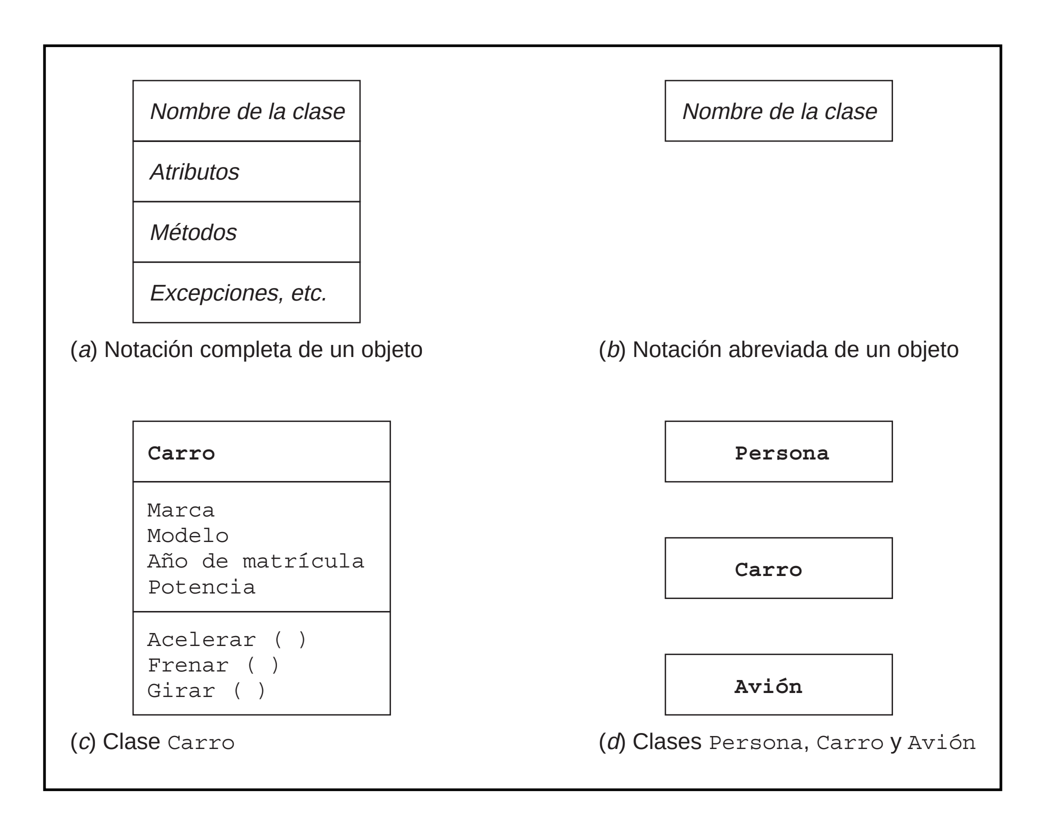


Figura 1.15. Representación de clases en UML.

Una clase es, por consiguiente, una descripción de un número de objetos similares. `Madonna`, `Sting`, `Prince`, `Juanes`, `Carlos Vives` o `Juan Luis Guerra` son miembros u objetos de la clase "músicos de rock". Un objeto concreto, `Juanes` o `Carlos Vives`, son *instancias* de la clase `"músicos de rock"`.

En **C++** una clase es una estructura de dato o tipo de dato que contiene funciones (métodos) como miembros y datos. Una clase es una descripción general de un conjunto de objetos similares. Por definición todos los objetos de una clase comparten los mismos atributos (datos) y las mismas operaciones (métodos). Una clase encapsula las abstracciones de datos y operaciones necesarias para describir una entidad u objeto del mundo real.

Una clase se representa en **UML** mediante un rectángulo que contiene en una banda con el nombre de la clase y opcionalmente otras dos bandas con el nombre de sus atributos y de sus operaciones o métodos (Figura 1.16).

1.6.6. Generalización y especialización: herencia

La *generalización* es la propiedad que permite compartir información entre dos entidades evitando la redundancia. En el comportamiento de objetos existen con frecuencia propiedades que son comunes en diferentes objetos y esta propiedad se denomina generalización.

Por ejemplo, máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas, etc., son todos electrodomésticos (aparatos del hogar). En el mundo de la orientación a objetos, cada uno de estos aparatos es un **`subclase`** de la clase `Electrodoméstico` y a su vez `Electrodoméstico` es una **`superclase`** de todas las otras clases `(máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas,...)`. El proceso inverso de la generalización por el cual se definen nuevas clases a partir de otras ya existentes se denomina *especialización*

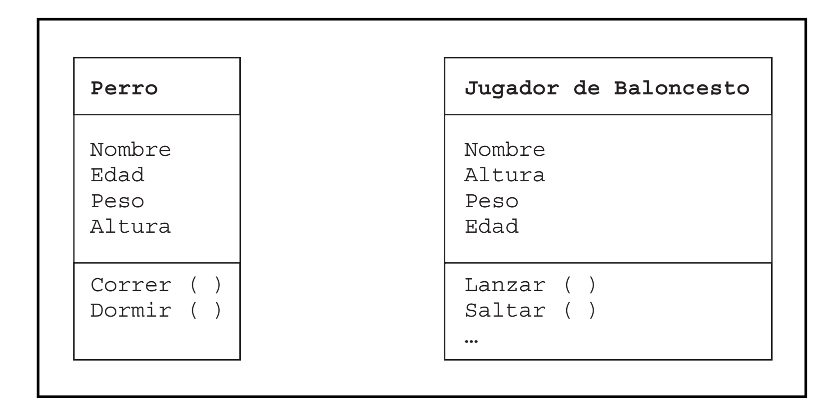


Figura 1.16. Representación de clases en UML con atributos y métodos.

En orientación a objetos, el mecanismo que implementa la propiedad de generalización se denomina **herencia.** La herencia permite definir nuevas clases a partir de otras clases ya existentes, de modo que presentan las mismas características y comportamiento de éstas, así como otras adicionales.

La idea de clases conduce a la idea de herencia. Clases diferentes se pueden conectar unas con otras de modo jerárquico. Como ya se ha comentado anteriormente con las relaciones de generalización y especialización, en nuestras vidas diarias se utiliza el concepto de clases divididas en subclases. La clase `animal` se divide en `anfibios`, `mamíferos`, `insectos`, `pájaros`, etc., y la clase `vehículo` en `carros`, `motos`, `camiones`, `buses`, etc.

El principio de la división o clasificación es que cada subclase comparte características comunes con la clase de la que procede o se deriva. Los carros, motos, camiones y buses tiene ruedas, motores y carrocerías; son las características que definen a un vehículo. Además de las características comunes con los otros miembros de la clase, cada subclase tiene sus propias características. Por ejemplo los camiones tienen una cabina independiente de la caja que transporta la carga; los buses tienen un gran número de asientos independientes para los viajeros que ha de transportar, etc. En la Figura 1.17 se muestran clases pertenecientes a una jerarquía o herencia de clases.

De modo similar una clase se puede convertir en padre o raíz de otras subclases**.** En C++ la clase original se denomina *clase base* y las clases que se derivan de ella se denominan *clases derivadas* y siempre son una especialización o *concreción* de su clase base. A la inversa, la clase base es la generalización de la clase derivada. Esto significa que todas las propiedades (atributos y operaciones) de la clase base se heredan por la clase derivada, normalmente suplementada con propiedades adicionales.

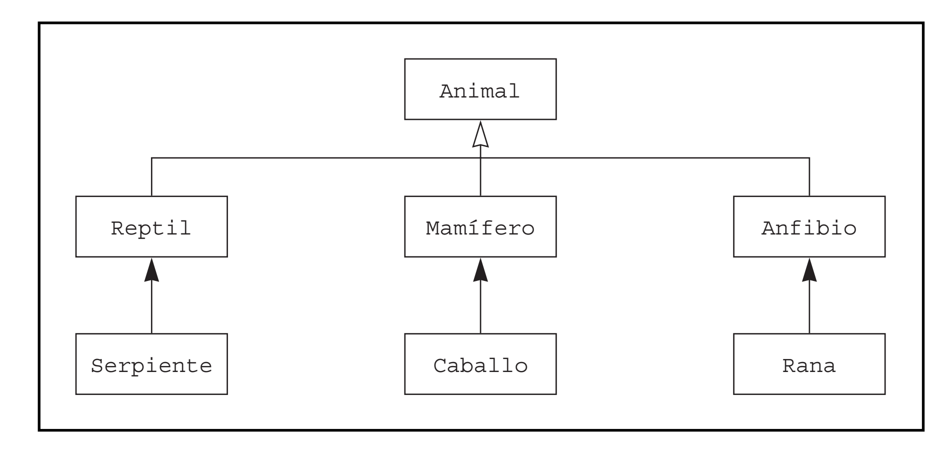


Figura 1.17. Herencia de clases en UML.

1.6.7 Reusabilidad

Una vez que una clase ha sido escrita, creada y depurada, se puede distribuir a otros programadores para utilizar en sus propios programas. Esta propiedad se llama *reusabilidad*[5] o *reutilización.* Su concepto es similar a las funciones incluidas en las bibliotecas de funciones de un lenguaje procedimental como C que se pueden incorporar en diferentes programas.

En C++, el concepto de herencia proporciona una extensión o ampliación al concepto de *reusabilidad.* Un programador puede considerar una clase existente y sin modificarla, añadir competencias y propiedades adicionales a ella. Esto se consigue derivando una nueva clase de una ya existente. La nueva clase heredará las características de la clase antigua, pero es libre de añadir nuevas características propias.

La facilidad de reutilizar o reusar el software existente es uno de los grandes beneficios de la POO: muchas empresas consiguen con la reutilización de clase en nuevos proyectos la reducción de los costes de inversión en sus presupuestos de programación. ¿En esencia cuales son las ventajas de la herencia? Primero, se utiliza para consistencia y reducir código. Las propiedades comunes de varias clases sólo necesitan ser implementadas una vez y sólo necesitan modificarse una vez si es necesario. La otra ventaja es que el concepto de abstracción de la funcionalidad común está soportada.

1.6.8. Polimorfismo

Además de las ventajas de consistencia y reducción de código, la herencia, aporta también otra gran ventaja: facilitar el polimorfismo. Polimorfismo es la propiedad de que un operador o una función actúen de modo diferente en función del objeto sobre el que se aplican. En la practica, el polimorfismo significa la capacidad de una operación de ser interpretada sólo por el propio objeto que lo invoca. Desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución ya que durante la compilación no se conoce qué tipo de objeto y por consiguiente que operación ha sido llamada. En el capítulo 14 se describirá en profundidad la propiedad de polimorfismo y los diferentes modos de implementación del polimorfismo.

La propiedad de **polimorfismo** es aquella en que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de diferentes formas en cada clase. Así, por ejemplo, la operación de abrir se puede dar en diferentes clases: abrir una puerta, abrir una ventana, abrir un periódico, abrir un archivo, abrir una cuenta corriente en un banco, abrir un libro, etc. En cada caso se ejecuta una operación diferente aunque tiene el mismo nombre en todos ellos "abrir". El polimorfismo es la propiedad de una operación de ser interpretada sólo por el objeto al que pertenece. Existen diferentes formas de implementar el polimorfismo y variará dependiendo del lenguaje de programación.

Veamos el concepto con ejemplos de la vida diaria.

En un taller de reparaciones de automóviles existen numerosos carros, de marcas diferentes, de modelos diferentes, de tipos diferentes, potencias diferentes, etc. Constituyen una clase o colección heterogénea de carros (coches). Supongamos que se ha de realizar una operación común "cambiar los frenos del carro". La operación a realizar es la misma, incluye los mismos principios, sin embargo, dependiendo del coche, en particular, la operación será muy diferente, incluirá diferentes acciones en cada caso. Otro ejemplo a considerar y relativo a los operadores "+" y "*" aplicados a números enteros o números complejos; aunque ambos son números, en un caso la suma y multiplicación son operaciones simples, mientras que en el caso de los números complejos al componerse de parte real y parte imaginaria, será necesario seguir un método específico para tratar ambas partes y obtener un resultado que también será un número complejo.

El uso de operadores o funciones de forma diferente, dependiendo de los objetos sobre los que están actuando se llama polimorfismo (una cosa con diferentes formas). Sin embargo, cuando un operador

[5] El término proviene del concepto ingles *reusability*. La traducción no ha sido aprobada por la RAE, pero se incorpora al texto por su gran uso y difusión entre los profesionales de la informática.

existente, tal como + o =, se le permite la posibilidad de operar sobre nuevos tipos de datos, se dice entonces que el operador está sobrecargado. La sobrecarga es un tipo de polimorfismo y una característica importante de la POO. En el Capítulo 10 se ampliará, también en profundidad, este nuevo concepto.

1.7. EL *SOFTWARE* (LOS PROGRAMAS)

El *software* de una computadora es un conjunto de instrucciones de programa detalladas que controlan y coordinan los componentes *hardware* de una computadora y controlan las operaciones de un sistema informático. El auge de las computadoras en el siglo pasado y en el actual siglo XXI, se debe esencialmente, al desarrollo de sucesivas generaciones de software potentes y cada vez más *amistosas* ("fáciles de utilizar").

Las operaciones que debe realizar el *hardware* son especificadas por una lista de instrucciones, llamadas programas, o *software*. Un programa de software es un conjunto de **sentencias** o **instrucciones** al computador. El proceso de escritura o codificación de un programa se denomina **programación** y las personas que se especializan en esta actividad se denominan **programadores**. Existen dos tipos importantes de software: software del sistema y software de aplicaciones. Cada tipo realiza una función diferente.

***Software* del sistema** es un conjunto generalizado de programas que gestiona los recursos del computador, tal como el procesador central, enlaces de comunicaciones y dispositivos periféricos. Los programadores que escriben software del sistema se llaman **programadores de sistemas**. ***Software* de aplicaciones** son el conjunto de programas escritos por empresas o usuarios individuales o en equipo y que instruyen a la computadora para que ejecute una tarea específica. Los programadores que escriben software de aplicaciones se llaman **programadores de aplicaciones**.

Los dos tipos de software están relacionados entre sí, de modo que los usuarios y los programadores pueden hacer así un uso eficiente del computador. En la Figura 1.18 se muestra una vista organizacional de un computador donde muestran los diferentes tipos de software a modo de capas de la computadora desde su interior (el hardware) hasta su exterior (usuario): Las diferentes capas funcionan gracias a las instrucciones específicas (instrucciones máquina) que forman parte del software del sistema y llegan al software de aplicación, programado por los programadores de aplicaciones, que es utilizado por el usuario que no requiere ser un especialista.

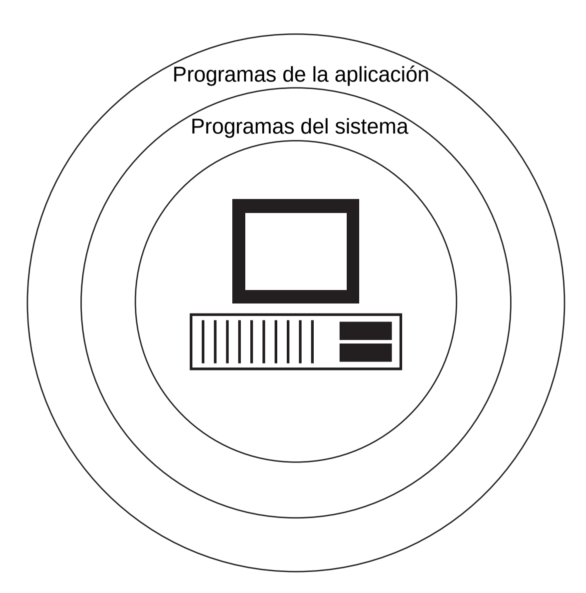


Figura 1.18. Relación entre programas de aplicación y programas del sistema.

1.7.1 Software del sistema

El software del sistema coordina las diferentes partes de un sistema de computadora y conecta e interactúa entre el software de aplicación y el hardware de la computadora. Otro tipo de software del sistema que gestiona controla las actividades de la computadora y realiza tareas de proceso comunes, se denomina *utility* o **utilidades** (en algunas partes de Latinoamérica, **utilerías**). El software del sistema que gestiona y controla las actividades del computador se denomina **sistema operativo**. Otro software del sistema son los programas traductores o de traducción de lenguajes de computador que convierten los lenguajes de programación, entendibles por los programadores, en lenguaje máquina que entienden las computadoras

El **software del sistema** es el conjunto de programas indispensables para que la máquina funcione; se denominan también *programas del sistema*. Estos programas son, básicamente, *el sistema operativo, los editores de texto,* los *compiladores/intérpretes* (lenguajes de programación) y los *programas de utilidad*.

1.7.2. Software de aplicación

El software de aplicación tiene como función principal asistir y ayudar a un usuario de un computador para ejecutar tareas específicas. Los programas de aplicación se pueden desarrollar con diferentes lenguajes y herramientas de software. Por ejemplo, una aplicación de procesamiento de textos (*word processing)* tal como Word o Word Perfect que ayuda a crear documentos, una hoja de cálculo tal como Lotus 1-2-3 o Excel que ayudan a automatizar tareas tediosas o repetitivas de cálculos matemáticos o estadísticos, a generar diagramas o gráficos, presentaciones visuales como PowerPoint, o a crear bases de datos como Acces u Oracle que ayudan a crear archivos y registros de datos.

Los usuarios, normalmente, compran el software de aplicaciones en discos CDs o DVDs (antiguamente en disquetes) o los descargan (bajan) de la Red Internet y han de instalar el software copiando los programas correspondientes de los discos en el disco duro de la computadora. Cuando compre estos programas asegúrese que son compatibles con su computador y con su sistema operativo. Existe una gran diversidad de programas de aplicación para todo tipo de actividades tanto de modo personal, como de negocios, navegación y manipulación en Internet, gráficos y presentaciones visuales, etc.

Los *lenguajes de programación* sirven para escribir programas que permitan la comunicación usuario/máquina. Unos programas especiales llamados *traductores* (**compiladores** o **intérpretes**) convierten las instrucciones escritas en lenguajes de programación en instrucciones escritas en lenguajes máquina (0 y 1, *bits*) que ésta pueda entender.

Los *programas de utilidad*[6] facilitan el uso de la computadora. Un buen ejemplo es un *editor de textos* que permite la escritura y edición de documentos. Este libro ha sido escrito en un editor de textos o *procesador de palabras* (“***word procesor***”).

Los programas que realizan tareas concretas, nóminas, contabilidad, análisis estadístico, etc., es decir, los programas que podrá escribir en Turbo Pascal, se denominan *programas de aplicación*. A lo largo del libro se verán pequeños programas de aplicación que muestran los principios de una buena programación de computadora.

Se debe diferenciar entre el acto de crear un programa y la acción de la computadora cuando ejecuta las instrucciones del programa. La creación de un programa se hace inicialmente en papel y, a continuación, se introduce en la computadora y se convierte en lenguaje entendible por la computadora. La Figura 1.19 muestra el proceso general de ejecución de un programa con una entrada (*datos*) al programa y la obtención de una salida (*resultados*). La entrada puede tener una variedad de formas, tales como texto, información numérica, imágenes o sonido. La salida puede también tener formas, tales como datos numéricos o caracteres, señales para controlar equipos o robots, etc.

---

[6] *Utility*: programa de utilidad o *utilería*.

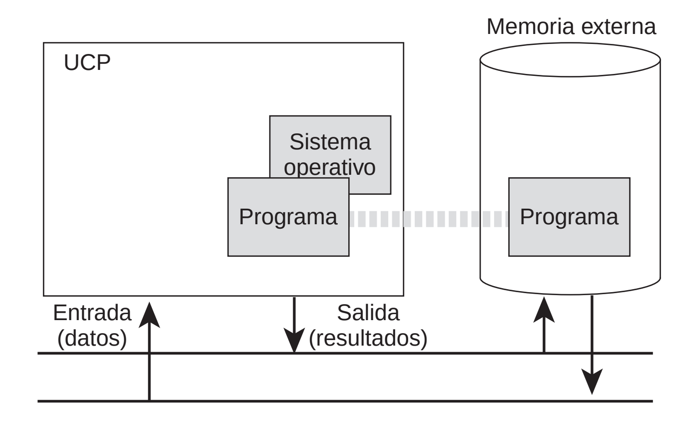


Figura 1.19. Ejecución de un programa.

1.8 SISTEMA OPERATIVO

Un sistema operativo **SO** (*Operating System, OS)* es tal vez la parte más importante del software del sistema y es el software que controla y gestiona los recursos del computador. En la práctica, el sistema operativo es la colección de programas de computador que controla la interacción del usuario y el hardware del computador. El sistema operativo es el administrador principal del computador, y por ello a veces, se le compara con el director de una orquesta ya que este software es el responsable de dirigir todas las operaciones del computador y gestionar todos sus recursos.

El sistema operativo asigna recursos, planifica el uso de recursos y tareas del computador, y monitoriza a las actividades del sistema informático. Estos recursos incluyen memoria, dispositivos de **E/S** (Entrada/Salida), y la **UCP** (**U**nidad **C**entral de **P**roceso). El sistema operativo proporciona servicios tales como asignar memoria a un programa y manipulación del control de los dispositivos de E/S tales como el monitor el teclado o las unidades de disco. La Tabla 1.6 muestra algunos de los sistemas operativos más populares utilizados en enseñanza y en informática profesional.

Cuando un usuario interactúa con un computador, la interacción está controlada por el sistema operativo. Un usuario se comunica con un sistema operativo a través de una interfaz de usuario de ese sistema operativo. Los sistemas operativos modernos utilizan una interfaz gráfica de usuario, **IGU** (*Graphical User Interface, GUI*) que hace uso masivo de iconos, botones, barras y cuadros de diálogo para realizar tareas que se controlan por el teclado o el ratón (mouse) entre otros dispositivos.

Normalmente, el sistema operativo se almacena de modo permanente en un chip de memoria de sólo lectura (ROM) de modo que esté disponible tan pronto el computador se pone en marcha ("se enciende" o "se prende"). Otra parte del sistema operativo puede residir en disco que se almacena en memoria RAM en la inicialización del sistema por primera vez en una operación que se llama c*arga* del sistema (*booting*).

Uno de los programas más importante es el **sistema operativo**, que sirve, esencialmente, para facilitar la escritura y uso de sus propios programas. El sistema operativo dirige las operaciones globales de la computadora, instruye a la computadora para ejecutar otros programas y controla el almacenamiento y recuperación de archivos (programas y datos) de cintas y discos. Gracias al sistema operativo es posible que el programador pueda introducir y grabar nuevos programas, así como instruir a la computadora para que los ejecute. Los sistemas operativos pueden ser: *monousuarios* (un solo usuario) y *multiusuarios*, o tiempo compartido (diferentes usuarios); atendiendo al número de usuarios y *monocarga* (una sola tarea) o *multitarea* (múltiples tareas) según las tareas (procesos) que puede realizar simultáneamente. C++ corre prácticamente en todos los sistemas operativos, Windows XP, Windows 95, Windows NT, Windows 2000, UNIX, Linux, Vista..., y en casi todas las computadoras personales actuales PC, Mac, Sun, etc.

Tabla 1.6. Sistemas operativos más utilizados en educación y en la empresa.

Sistema operativo	Características
Windows Vista[7]	Nuevo sistema operativo de Microsoft presentado en 2006, pero que se lanzará comercialmente en 2007.
Windows XP	Sistema operativo más utilizado en la actualidad, tanto en el campo de la enseñanza, como en la industria y negocios. Su fabricante es Microsoft.
Windows 98/ME/2000	Versiones anteriores de Windows pero que todavía hoy son muy utilizados.
UNIX	Sistema operativo abierto, escrito en C y todavía muy utilizado en el campo profesional.
Linux	Sistema operativo de software abierto, gratuito y de libre distribución, similar a UNIX, y una gran alternativa a Windows. Muy utilizado actualmente en servidores de aplicaciones para Internet.
Mac OS	Sistema operativo de las computadoras Apple Macintosh.
DOS y OS/2	Sistemas operativos creados por Microsoft e IBM respectivamente, ya poco utilizados pero que han sido la base de los actuales sistemas operativos.
CP/M	Sistema operativo de 8 bits para las primeras microcomputadoras nacidas en la década de los setenta.
Symbian	Sistema operativo para teléfonos móviles apoyado fundamentalmente por el fabricante de teléfonos celulares Nokia.
PalmOS	Sistema operativo para agendas digitales, PDA, del fabricante Palm.
Windows Mobile, CE	Sistema operativo para teléfonos móviles con arquitectura y apariencias similares a Windows XP.

1.8.1. Tipos de sistemas operativos

Las diferentes características especializadas del sistema operativo permiten a los computadores manejar muchas diferentes tareas así como múltiples usuarios de modo simultáneo o en paralelo, bien de modo secuencial... En base a sus características específicas los sistemas operativos se pueden clasificar en varios grupos:

Multiprogramación/Multitarea

La multiprogramación permite a múltiples programas compartir recursos de un sistema de computadora en cualquier momento a través del uso concurrente de una UCP. Sólo un programa utiliza realmente la UCCP en cualquier momento dado, sin embargo, las necesidades de entrada/salida pueden ser atendidas en el mismo momento. Dos o más programas están activos al mismo tiempo, pero no utilizan los recursos del computador simultáneamente. Con multiprogramación, un grupo de programas se ejecutan alternativamente y se alternan en el uso del procesador. Cuando se utiliza un sistema operativo de un único usuario, la multiprogramación toma el nombre de **multitarea**.

Multiprogramación

Método de ejecución de dos o más programas concurrentemente utilizando la misma computadora. La UCO ejecuta sólo un programa pero puede atender los servicios de entrada/salida de los otros al mismo tiempo.

[7] Microsoft tiene previsto presentar en el año 2006, un nuevo sistema operativo llamado Windows Vista, actualización de Windows XP pero con numerosas funcionalidades, especialmente de Internet y de seguridad, incluyendo en el sistema operativo programas que actualmente se comercializan independientes, tales como programas de reproducción de música, vídeo, y fundamentalmente un sistema de representación gráfica muy potente que permitirá construir aplicaciones en tres dimensiones, así como un buscador, un sistema antivirus y otras funcionalidades importantes.

Tiempo compartido (múltiples usuarios, time sharing)

Un sistema operativo multiusuario es un sistema operativo que tiene la capacidad de permitir que muchos usuarios compartan simultáneamente los recursos de proceso de la computadora. Centenas o millares de usuarios se pueden conectar al computador que asigna un tiempo de computador a cada usuario, de modo que a medida que se libera la tarea de un usuario, se realiza la tarea del siguiente, y así sucesivamente. Dada la alta velocidad de transferencia de las operaciones, la sensación es de que todos los usuarios están conectados simultáneamente a la UCP con cada usuario recibiendo únicamente un tiempo de máquina.

Multiproceso

Un sistema operativo trabaja en multiproceso cuando puede enlazar a dos o más UCPs para trabajar en paralelo en un único sistema de computadora. El sistema operativo puede asignar múltiples UCPs para ejecutar diferentes instrucciones del mismo programa o de programas diferentes simultáneamente, dividiendo el trabajo entre las diferentes UCP.

La multiprogramación utiliza proceso concurrente con una CPU; el multiproceso utiliza proceso simultáneo con múltiples CPUs.

1.9. LENGUAJES DE PROGRAMACIÓN

Como se ha visto en el apartado anterior, para que un procesador realice un proceso se le debe suministrar en primer lugar un algoritmo adecuado. El procesador debe ser capaz de *interpretar* el algoritmo, lo que significa:

- comprender las instrucciones de cada paso,
- realizar las operaciones correspondientes.

Cuando el procesador es una computadora, el algoritmo se ha de expresar en un formato que se denomina *programa*, ya que el pseudocódigo o el diagrama de flujo no son comprensibles por la computadora, aunque pueda entenderlos cualquier programador. Un programa se escribe en un *lenguaje de programación* y las operaciones que conducen a expresar un algoritmo en forma de programa se llaman *programación*. Así pues, los lenguajes utilizados para escribir programas de computadoras son los *lenguajes de programación* y **programadores** son los escritores y diseñadores de programas. El proceso de traducir un algoritmo en *pseudocódigo* a un lenguaje de programación se denomina **codificació**n, y el algoritmo escrito en un lenguaje de programación se denomina **código fuente**.

En la realidad la computadora no entiende directamente los lenguajes de programación sino que se requiere un programa que traduzca el código fuente a otro lenguaje que sí entiende la máquina directamente, pero muy complejo para las personas; este lenguaje se conoce como **lenguaje máquina** y el código correspondiente **código máquina**. Los programas que traducen el código fuente escrito en un lenguaje de programación —tal como C++— a código máquina se denominan **traductores**. El proceso de conversión de un algoritmo escrito en pseudocódigo hasta un programa ejecutable comprensible por la máquina, se muestra en la Figura 1.20.

Hoy en día, la mayoría de los programadores emplean lenguajes de programación como C++, C, C#, Java, Visual Basic, XML, HTML, Perl, PHP, JavaScript..., aunque todavía se utilizan, sobre todo profesionalmente, los clásicos *COBOL*, *FORTRAN*, *Pascal* o el mítico BASIC. Estos lenguajes se denominan **lenguajes de alto nivel** y permiten a los profesionales resolver problemas convirtiendo sus algoritmos en programas escritos en alguno de estos lenguajes de programación.

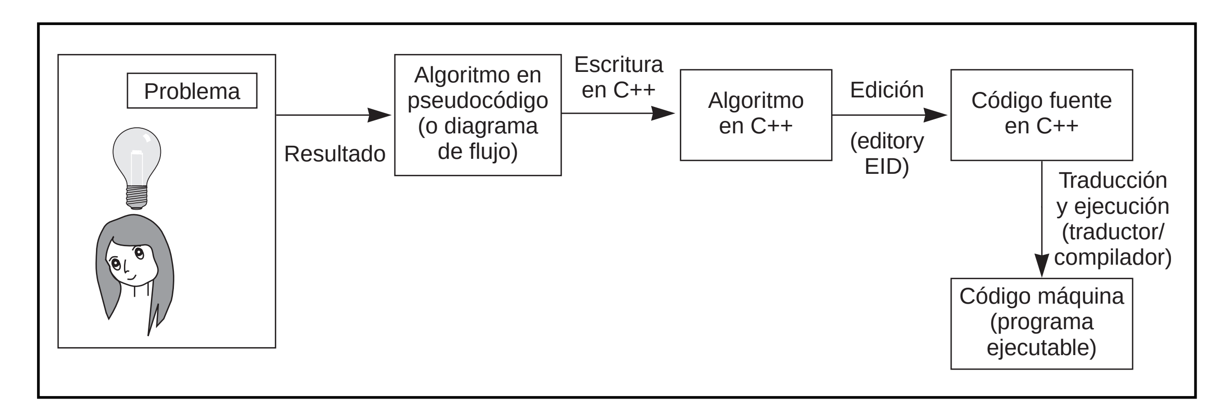


Figura 1.20. Proceso de transformación de un algoritmo en pseudocódigo en un programa ejecutable.

1.9.1. Traductores de lenguaje: el proceso de traducción de un programa

El proceso de traducción de un programa fuente escrito en un lenguaje de alto nivel a un lenguaje máquina comprensible por la computadora, se realiza mediante programas llamados "traductores". Los **traductores de lenguaje** son programas que traducen a su vez los programas fuente escritos en lenguajes de alto nivel a código máquina. Los traductores se dividen en **compiladores** e **interpretes**.

Intérpretes

Un *intérprete* es un traductor que toma un programa fuente, lo traduce y, a continuación, lo ejecuta. Los programas intérpretes clásicos como BASIC, prácticamente ya no se utilizan, más que en circunstancias especiales. Sin embargo, está muy extendida la versión interpretada del lenguaje Smalltalk, un lenguaje orientado a objetos puro. El sistema de traducción consiste en: traducir la primera sentencia del programa a lenguaje máquina, se detiene la traducción, se ejecuta la sentencia; a continuación, se traduce la siguiente sentencia, se detiene la traducción, se ejecuta la sentencia y así sucesivamente hasta terminar el programa.

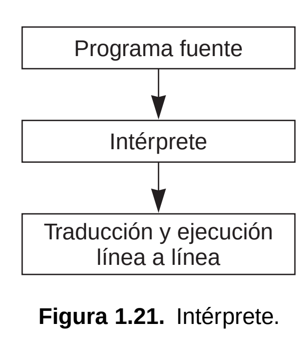


Figura 1.21. Intérprete.

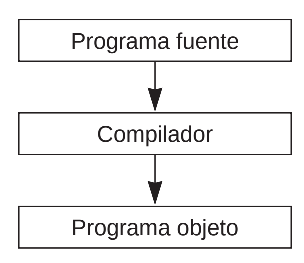


Figura 1.22. La compilación de programas.

Compiladores

Un *compilador* es un programa que traduce los programas fuente escritos en lenguaje de alto nivel a lenguaje máquina. La traducción del programa completo se realiza en una sola operación denominada **compilación** del programa; es decir, se traducen todas las instrucciones del programa en un solo bloque. El programa compilado y depurado (eliminados los errores del código fuente) se denomina *programa ejecutable* porque ya se puede ejecutar directamente y cuantas veces se desee; sólo deberá volver a compilarse de nuevo en el caso de que se modifique alguna instrucción del programa. De este modo el programa ejecutable no necesita del compilador para su ejecución. Los lenguajes compiladores típicos más utilizados son: **C**, **C++**, **Java**, **C#**, **Pascal**, **FORTRAN** y **COBOL**.

1.9.2. La compilación y sus fases

La *compilación* es el proceso de traducción de programas fuente a programas objeto. El programa objeto obtenido de la compilación ha sido traducido normalmente a código máquina.

Para conseguir el programa máquina real se debe utilizar un programa llamado *montador* o *enlazador* (*linker*). El proceso de montaje conduce a un programa en lenguaje máquina directamente ejecutable (Figura 1.23).

El proceso de ejecución de un programa escrito en un lenguaje de programación y mediante un compilador suele tener los siguientes pasos:

1. Escritura del *programa fuente* con un *editor* (programa que permite a una computadora actuar de modo similar a una máquina de escribir electrónica) y guardarlo en un dispositivo de almacenamiento (por ejemplo, un disco).
2. Introducir el programa fuente en memoria.
3. *Compilar* el programa con el compilador C.
4. *Verificar* y *corregir errores de compilación* (listado de errores).
5. Obtención del programa *objeto*.
6. El enlazador (*linker*) obtiene el *programa ejecutable*.
7. Se ejecuta el programa y, si no existen errores, se tendrá la salida del programa.

El proceso de ejecución sería el mostrado en las figuras 1.24 y 1.25.

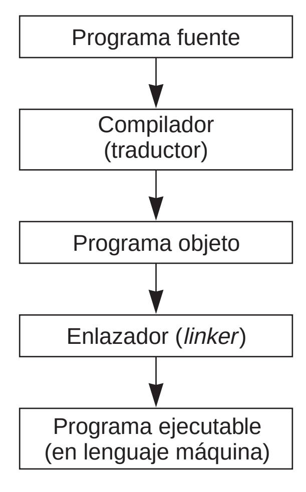


Figura 1.23. Fases de la compilación.

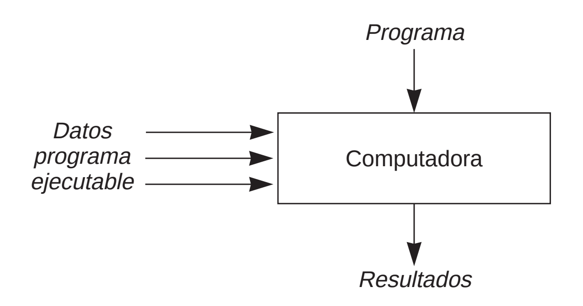


Figura 1.24. Ejecución de un programa.

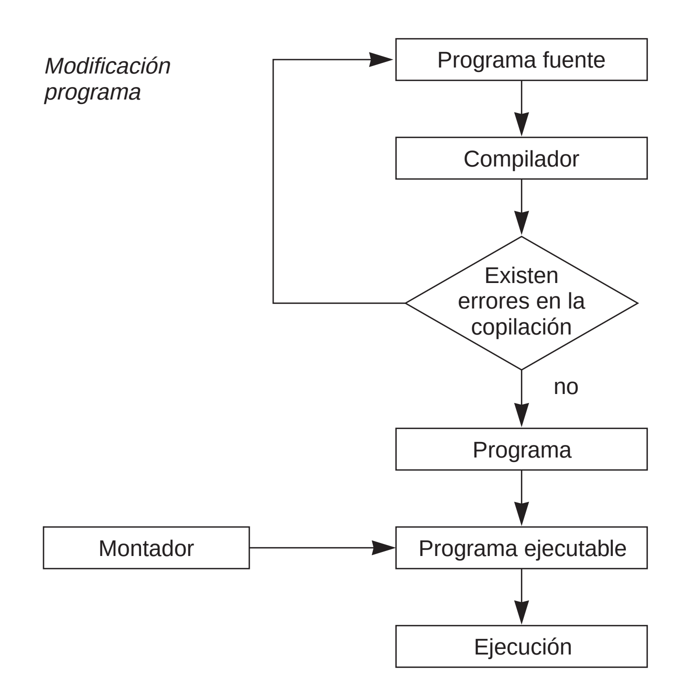


Figura 1.25. Fases de ejecución de un programa.

En el Capítulo 2 se describirá en detalle el proceso completo y específico de ejecución de programas en lenguaje C.

1.10. C: EL ORIGEN DE C++ COMO LENGUAJE UNIVERSAL

C es el lenguaje de programación de propósito general asociado, de modo universal, al sistema operativo UNIX. Sin embargo, la popularidad, eficacia y potencia de C, se ha producido porque este lenguaje no está prácticamente asociado a ningún sistema operativo, ni a ninguna máquina, en especial. Ésta es la razón fundamental, por la cual C, es conocido como el ***lenguaje de programación de sistemas, por excelencia***.

C es una evolución de los lenguajes BCPL —desarrollado por Martin Richards— y B —desarrollado por Ken Thompson en 1970— para el primitivo INIX de la computadora DEC PDP-7.

C nació realmente en 1978, con la publicación de *The C Programming Languaje*, por Brian Kernighan y Dennis Ritchie (Prentice Hall, 1978). Desde su nacimiento, C fue creciendo en popularidad y los

sucesivos cambios en el lenguaje a lo largo de los años junto a la creación de compiladores por grupos no involucrados en su diseño, hicieron necesario pensar en la estandarización de la definición del lenguaje C.

Así, en 1983, el American National Standard Institute (ANSI), una organización internacional de estandarización, creó un comité (el denominado X3J11) cuya tarea fundamental consistía en hacer *"una definición no ambigua del lenguaje C, e independiente de la máquina"*. Había nacido el estándar ANSI del lenguaje C. Con esta definición de C se asegura que cualquier fabricante de software que vende un compilador ANSI C incorpora todas las características del lenguaje, especificadas por el estándar. Esto significa también que los programadores que escriban programas en C estándar tendrán la seguridad de que correrán sus modificaciones en cualquier sistema que tenga un compilador C.

C es un *lenguaje de alto nivel*, que permite programar con instrucciones de lenguaje de propósito general. También, C se define como un lenguaje de programación estructurado de propósito general; aunque en su diseño también primó el hecho que fuera especificado como un lenguaje de programación de Sistemas, lo que proporciona una enorme cantidad de potencia y flexibilidad.

El estándar ANSI C formaliza construcciones no propuestas en la primera versión de C, en especial, asignación de estructuras y enumeraciones. Entre otras aportaciones, se definió esencialmente, una nueva forma de declaración de funciones (prototipos). Pero es, esencialmente, la biblioteca estándar de funciones, otra de las grandes aportaciones.

Hoy, en el siglo XXI, C sigue siendo uno de los lenguajes de programación más utilizados en la industria del software, así como en institutos tecnológicos, escuelas de ingeniería y universidades. Prácticamente todos los fabricantes de sistemas operativos, Windows, UNIX, Linux, MacOS, Solaris..., soportan diferentes tipos de compiladores de lenguaje C y en muchas ocasiones distribuciones gratuitas bajo cualquiera de los sistemas operativos citados. Todos los compiladores de C++ pueden ejecutar programas escritos en lenguaje C, preferentemente si cumplen el estándar ANSI C.[8]

1.11. EL LENGUAJE C++: HISTORIA Y CARACTERÍSTICAS

C++, Java y C#, los tres lenguajes más populares junto con C en esta primera década del siglo XXI son herederos directos del propio C con características orientadas a objetos y a Internet. Actualmente, y aunque C sigue siendo, tal vez, el más utilizado en el mundo de la educación como primer lenguaje de programación y también copa un porcentaje alto de utilización en el campo profesional, los tres lenguajes con características técnicas de orientación a objetos forman con C el *poker* de lenguajes más empleados en el mundo educativo, profesional y científico actual y previsiblemente de los próximos años.

C++ es heredero directo del lenguaje C que, a su vez, se deriva del lenguaje B [Richards, 1980]. C se mantiene como un subconjunto de C++. Otra fuente de inspiración, como señala su autor Bjarne Stroustrup [Stroustrup, 1997][9] fue Simula 67 [Dahl, 1972] del que tomó el concepto de clase (con clases derivadas y funciones virtuales).

El lenguaje de programación C fue desarrollado por **Dennis Ritche** de AT&T Bell Laboratories que se utilizó para escribir y mantener el sistema operativo UNIX (hasta que apareció C, el sistema operativo UNIX fue desarrollado por **Ken Thompson** en AT&T Bell Laboratories mediante en lenguaje ensamblador o en B). C es un lenguaje de propósito general que se puede utilizar para escribir cualquier tipo de programa, pero su éxito y popularidad está especialmente relacionado con el sistema operativo UNIX. (Fue desarrollado como *lenguaje de programación de sistemas*, es decir, un lenguaje de programación para escribir *sistemas operativos* y utilidades (programas) del sistema.) Los sistemas operativos

---

[8] Opciones gratuitas buenas puede encontrar en el sitio del fabricante de software Borland. También puede encontrar y descargar un compilador excelente Dev-C++ en software libre que puede compilar código C y también código C++, en `www.bloodshed.net` y en `www.download.com` puede asímismo encontrar diferentes compiladores totalmente gratuitos. Otros numerosos sitios puede encontrar en software gratuito en numerosos sitios de la red. Los fabricantes de software y de computadoras (IBM, Microsoft, HP...) ofrecen versiones a sus clientes aunque normalmente no son gratuitos

[9] P. 11

son los programas que gestionan (administran) los *recursos de la computadora*. Ejemplos bien conocidos de sistemas operativos además de UNIX son MS/DOS, OS/2, MVS, Lynux, Windows 95/98, Windows NT, Windows 2000, OS Mac, etc.

La especificación formal del lenguaje C es un documento escrito por Ritchie, titulado *The C Reference Manual*. En 1997, **Ritchie** y **Brian Kernighan**, ampliaron ese documento y publicaron un libro referencia del lenguaje *The C Programming Language* (también conocido por el K&R).

Aunque C es un lenguaje muy potente, tiene dos características que lo hacen inapropiado como una introducción moderna a la programación. Primero, C requiere un nivel de sofisticación a sus usuarios que les obliga a un difícil aprendizaje a los programadores principiantes ya que es de comprensión difícil. Segundo C, fue diseñado al principio de los setenta, y la naturaleza de la programación ha cambiado de modo significativo en la década de los ochenta y noventa.

Para subsanar estas "deficiencias" Bjarne Stroustrup de AT&T Bell Laboratories desarrolló C++ al principio de la década de los ochenta. Stroustrup diseñó C++ como un mejor C. En general, C estándar es un subconjunto de C++ y la mayoría de los programas C son también programas C++ (la afirmación inversa no es verdadera). C++ además de añadir propiedades a C, presenta características y propiedades de *programación orientada a objetos*, que es una técnica de programación muy potente y que se verá en la última parte de este libro.

Se han presentado varias versiones de C++ y su evolución se estudió en [Stroustrup 94]. Las características más notables que han ido incorporándose a C++ son: herencia múltiple, *genericidad*, plantillas, funciones virtuales, excepciones, etc. C++ ha ido evolucionando año a año y como su autor ha explicado: "*evolucionó siempre para resolver problemas encontrados por los usuarios y como consecuencia de conversaciones entre el autor, sus amigos y sus colegas*"[10].

IMPORTANTE: Página oficial de Bjarne Stroustrup

Bjarne Stroustrupo, diseñador e implementador del lenguaje de programación C++ es la referencia fundamental y definitiva para cualquier estudiante y programador de C++. Sus obras *The C++ Programming Language*[11], *The Design and Evolution of C++ y C++. Reference Manual* son lectura y consulta obligada.

Su sitio web personal de AT&T Labs Researchs debe ser el primer sitio "favorito" que le recomendamos visite con cierta frecuencia.

www.resarch.att.com/~bs

El sitio es actualizado con frecuencia por Stroustrup y contiene gran cantidad de información y una excelente sección de FAQ (*frequently asked questions*).

C++ comenzó su proyecto de estandarización ante el comité ANSI y su primera referencia es *The Annotated C++ Reference Manual* [**Ellis** 89][12]. En diciembre de 1989 se reunió el comité X3J16 del ANSI por iniciativa de Hewlett Packard. En junio de 1991, a la estandarización de ANSI se unió ISO (*International Organization for Standardization*) con su propio comité (ISO-WG-21), creando un esfuerzo común ANSI/ISO para desarrollar un estándar para C++. Estos comités se reunen tres veces al año para aunar sus esfuerzos y llegar a una decisión de creación de un estándar que convirtiera a C++ en un lenguaje importante y de amplia difusión.

En 1995, el Comité publicó diferentes artículos (*working paper*) y en abril de ese año, se publicó un borrador del estándar (Comité Draft) para su examen público. En diciembre de 1996 se lanzó una segunda versión (CD2) a dominio público. Estos documentos no sólo refinaron la descripción de las caracte-

---

[10] [Stroustrup 98], p. 12

[11] Esta obra ha sido traducida por un equipo de profesores de la universidad pontificia de Salamanca que coordinó y dirigió el autor de este libro.

[12] Existe versión española de Addison-Wesley Díaz de Santos y traducida por los profesores Manuel Katrib y Luis Joyanes.

rísticas existentes, de C++ sino que también se amplió el lenguaje con excepciones, identificación en tiempo de ejecución (*RTII*, *run_time type identification*), plantillas (templates) y la biblioteca estándar de plantillas STL (*Standard Template Library*). Stroustrup publicó en 1997 la tercera edición de su libro *The C++ Programming Language*. Este libro sigue el estándar ANSI/ISO C++.

C++ es un lenguaje estandarizado. En 1998, un comité de ANSI/ISO (*American Nacional Standard Institute/International Organization for Standardization*) adoptó una especificación del lenguaje conocida como Estándar C++ de 1998, oficialmente tiene el título ISO/ANSI C++ Standard (ISO/IEC 14882:1998) [Standard98]. En 2003 una segunda versión del Estándar vio la luz y fue adoptada también como [Standard03]. El estándar está disponible en Internet como archivo PDF con el número de documento 14882 en *http://www.ansi.org* donde se puede adquirir. La nueva edición es una revisión técnica, significando que se ordena la primera edición —fijación de tipos, reducción de ambigüedades y similares, pero no cambian las características del lenguaje. Este libro se basa en el estándar C++ como un superconjunto válido de C. Existen diferencias entre el estándar ANSI C y las reglas correspondientes de C++, pero son pocas. Realmente, ANSI C incorpora algunas características primitivas introducidas en C++, tal como el prototipado de funciones y el calificador de tipo `const`.

Antes de la emergencia de ANSI C, la comunidad de C fue seguida de un estándar de facto, basado en el libro *The C Programming Language*, de Kernighan y ritchie (Addison-Wesley Publishing Company, Reading, MA, 1978). Este estándar se conoció, en un principio, como K&r; con la emergencia de ANSI C, el k&R más simple se llamó, con frecuencia, C clásico.

El estándar ANSI C no sólo definió el lenguaje C sino que también definió una biblioteca estándar de C que deben soportar todas las implementaciones de ANSI C. C++ también utiliza la biblioteca; este libro se refiere a ella como la biblioteca estándar de C o simplemente *biblioteca estándar*. También el estándar ANSI/ISO C++ proporciona una biblioteca estándar de clases.

Recientemente, el C estándar ha sido revisado; el nuevo estándar, llamado, con frecuencia C99, fue adoptado por ISO en 1999 y por ANSI en 2000. Este estándar añade algunas características de C, tales como un nuevo tipo integer, que soportan algunos compiladores de C++. Aunque, no son parte del estándar C++ actual, estas características se pueden convertir como parte del C++ Standard. Antes de que el comité ANSI/ISO C++ comenzara su trabajo, muchas personas aceptaron como estándar, la versión de C++ más reciente de Bell Labs. Así, un compilador puede describirse como compatible con la versión 2.0 o 3.0 de C++.

Debido a que se lleva mucho tiempo hasta que un fabricante de compiladores implementa las últimas especificaciones de un lenguaje, existen compiladores que todavía no cumplen o conforman el estándar. Esto puede conducir a restricciones en algunos casos. Sin embargo, todos los programas de este libro han sido compilados con las últimas versiones de compiladores diferentes.

C++ es un lenguaje orientado a objetos que se ha hecho muy popular y con una gran difusión en el mundo del software y actualmente constituye un lenguaje estándar para programación orientada a objetos, aunque también puede ser utilizado como lenguaje estructurado al estilo de C si se desea trabajar al modo clásico de programación estructurada, sobre todo si se desea trabajar con algoritmos y estructura de datos.

1.11.1. C versus C++

C++ es una extensión de C con características más potentes. Estrictamente hablando, es un superconjunto de C. Al igual que sucede con Java y C# que son superconjuntos de C++. El ANSI C estándar no sólo define el lenguaje C sino que también define una biblioteca de C estándar que las implementaciones de ANSI C deben soportar. C++ también utiliza esa biblioteca, además de su propia biblioteca estándar de clases. Hace unos años se revisó el nuevo estándar de C, denominado C99 que fue adoptado por ISO en 1999 y por ANSI en 2000. El estándar de C ha añadido algunas características que soportan algunos compiladores de C++.

Por estas razones casi todas las sentencias de C también tienen una sentencia correcta en C++, pero no es cierto a la inversa. Los elementos más importantes añadidos a C para crear clases C, objetos y

programación orientada a objetos (C++ fue llamado originalmente "C con clases"). Sin embargo, a C++ se han añadido nuevas características, incluyendo un enfoque mejorado de la entrada/salida (E/S) y un nuevo medio para escribir comentarios. La Figura 1.26 muestra las relaciones entre C y C++

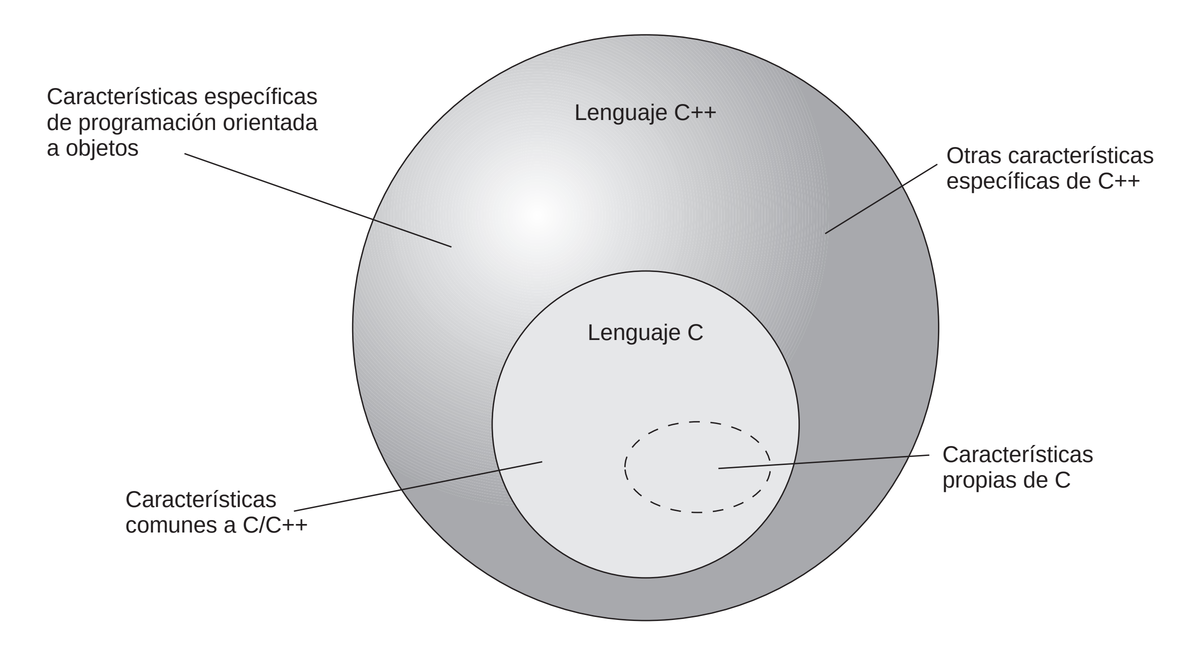


Figura 1.26. Características de C y C++.

De hecho las diferencias prácticas entre C y C+ son mucho mayores que lo que se pueda pensar. Aunque se puede escribir un programa en C++ similar a un programa en C, raramente se hace. C++ proporciona a los programadores las nuevas características de C++ y enfatiza a utilizar estas características, pero también se suele utilizar en partes de los programas las características específicas y potentes de C. Si usted ya conoce C, muchas propiedades le serán familiares y aprenderá de un modo mucho más rápido y eficiente, pero encontrará muchas propiedades nuevas y muy potentes que le harán disfrutar con este nuevo lenguaje y construir sus programas aprovechando la potencia de orientación a objetos y genérica de C++.

Nuestro objetivo es ayudarle a escribir programas POO tan pronto como sea posible, Sin embargo, como ya se ha observado, muchas características de C++ se han heredado de C, de modo que, aunque la estructura global de un programa pueda ser POO, consideramos que usted necesita conocimientos profundos del "viejo estilo *procedimental*". Por ello, los capítulos de la primera parte del libro le van introduciendo lenta y pausadamente en las potentes propiedades orientadas a objetos de las últimas partes, al objeto de conseguir a la terminación del libro el dominio de la Programación en C++.

Este libro describe el estándar ANSI/ISO C++, segunda edición, (ISO/IEC 14882:2003), de modo que los ejemplos funcionarán con cualquier implementación de C++ que sea compatible con el estándar. Sin embargo, pese a los años transcurridos, C++ Standard se suele considerar todavía nuevo y se pueden encontrar algunas discrepancias con el compilador que esté usted utilizando. Por ejemplo, si su compilador no es una versión reciente, puede carecer de espacios de nombres o de las características más novedosas de plantillas, como la clase `"String"` y la *Standard Template Library* que no son compatibles con los compiladores más antiguos. También puede suceder que si un compilador no cumple el estándar, algunas propiedades tales como el número de bytes usados para contener un entero, son dependientes de la implementación.

1.11.2 El futuro de C++

C++ continúa evolucionando y se habían realizado ya trabajos para producir la próxima versión del estándar. La nueva versión se ha llamado de modo informal como **C++0X**, ya que se espera su terminación al final de esta década, alrededor de 2009. Bjarne Stroustrup ha publicado el 2 de enero de 2006, un artículo titulado *The Sc++ Source: A Brief Look at C++0X* donde plantea brevemente los principios del trabajo del nuevo estándar y las esperanzas que existen para que el comité ISO C++ termine sus trabajos en 2008 y se puede convertir de facto en el nuevo estándar **C++09**. En otras palabras, comenta Struoustrup, se trata de reforzar el estándar **C++98** que ya es un lenguaje fuertemente implantado. Se pretende hacer de C++ un lenguaje mejor para programación de sistemas y construcción de bibliotecas, así como un lenguaje C++ que sea más fácil de enseñar y de aprender.

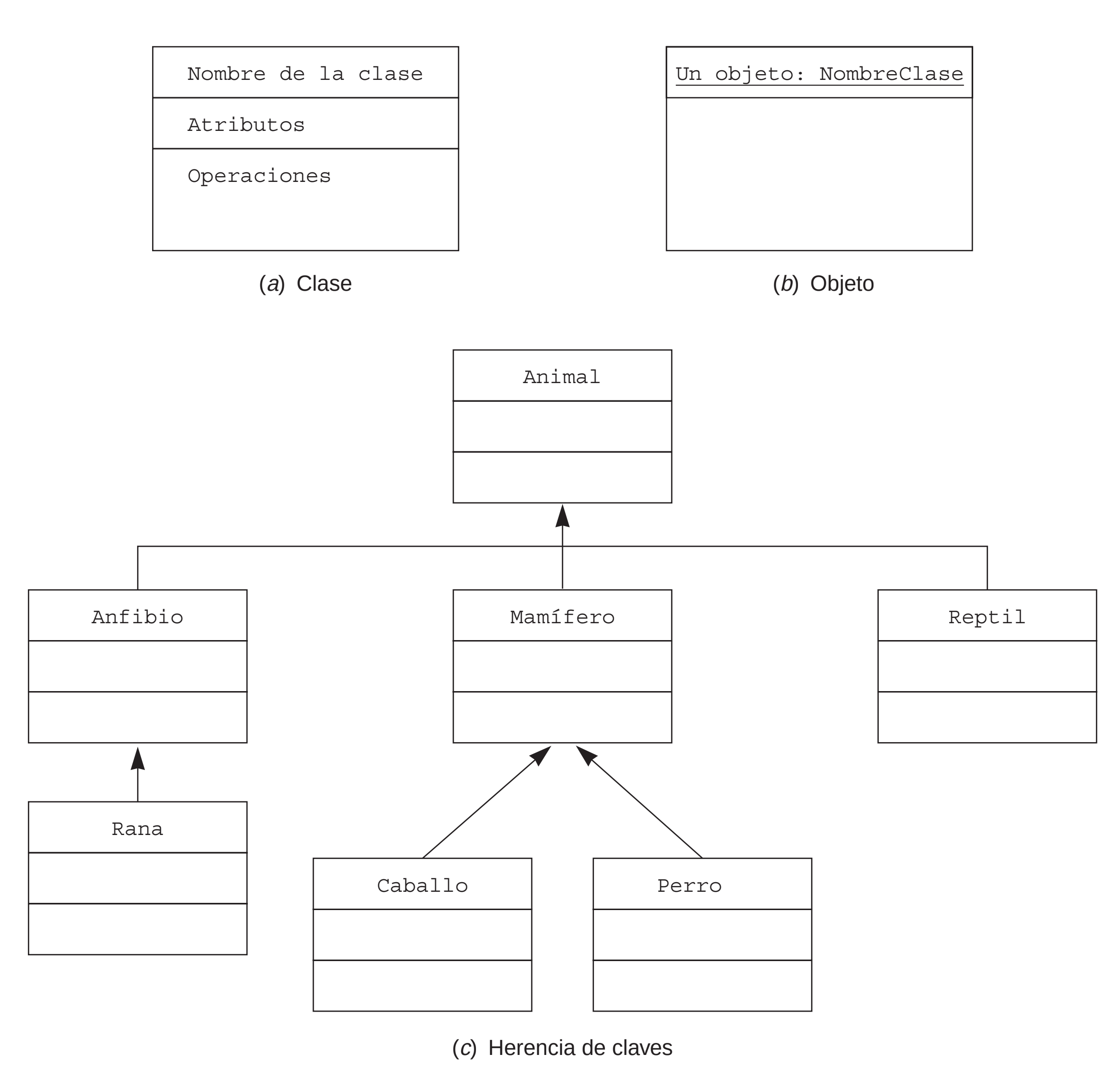
1.12. EL LENGUAJE UNIFICADO DE MODELADO (UML 2.0)

UML es un lenguaje gráfico para modelado de programas de computadoras. "Modelado" significa como su nombre indica, crear modelos o representaciones de "algo", como un plano de una casa o similar. UML proporciona un medio de visualizar la organización de alto nivel de los programas sin fijarse con detenimiento en los detalles del código real.

El lenguaje de modelado está unificado porque está basado en varios modelos previos (métodos de Booch, Rumbaugh y Jacobson). En la actualidad UML está adoptado por OMG (*Object Management Group*), un consorcio de más de 1.000 sociedades y universidades activas en el campo de tecnologías orientadas a objetos y está dedicado especialmente a unificación de estándares. UML tiene una gran riqueza semántica que lo abstrae de numerosos aspectos técnicos y ésa es su gran fortaleza. ¿Porqué necesitamos UML en programación orientada a objetos? En primer lugar, porqué UML nació como herramientas gráficas y metodologías para implementar análisis y diseño orientados a objetos y, en segundo lugar, porque en programas grandes de computadoras es difícil entender el funcionamiento de un programa examinando sólo su código y es necesario ver cómo se relacionan unas partes con otras.

La parte más importante de UML, al menos a un nivel de iniciación o medio en programación, reside en un conjunto rico de diagramas gráficos. Diagramas de clases que muestran las relaciones entre clases, diagramas de objetos que muestran cómo se relaciona objetos específicos entre sí, diagramas de casos de uso que muestran cómo los usuarios de un programa interactúan con el programa, etc. Cuando se modelan clases, UML es de hecho estándar para representaciones gráficas. UML no es un proceso de desarrollo de software sino, simplemente, un medio para examinar el software que se está desarrollando. Aunque, al ser un estándar, UML se puede aplicar a cualquier tipo de lenguaje de programación, está especialmente adaptado a la POO. En la Figura 1.27 se muestran representaciones gráficas de clases, objetos y relaciones de generalización y especialización (herencia) entre clases.

Una breve reseña de UML es la siguiente. En 1994, James Rumbaugh y Grady Booch deciden unirse para unificar sus notaciones y métodos: OMT y Booch. En 1995, Juar Jacobson se une al equipo de «los tres amigos» en la empresa Rational Software. En 1997 se publica UML 1.0 y en ese mismo año OMG adopta la notación y crea una comisión (*Tark Forcs*) encargada de estudiar la evolución de UML. En 2003 se presenta UML 1.5. A lo largo de 2005 se presenta UML 2.0 y en 2006 Rumbaugh, Booch y Jacobson publican las nuevas ediciones de la Guía de Usuario y Manual de Referencia.



(*a*) Clase

(*b*) Objeto

(*c*) Herencia de claves

Figura 1.27. Representaciones gráficas de objetos, clases y herencia en UML 2.0.

RESUMEN

En este capítulo se realiza una breve descripción de la organización física de una computadora. Se describen los conceptos fundamentales de la misma y sus bloques y componentes más sobresalientes, tales como CPU (UCP), ALU (UAL), VC, microprocesador, memorias RAM, ROM, dispositivos de E/S y de almacenamiento (cintas, disquetes, discos duros, CDs, DVDs, memorias *flash* con y sin conexiones USB, etc.).

Se hace también una breve introducción a la representación de la información en textos, imágenes, sonidos y datos numéricos. Asimismo, se analiza una primera introducción a la programación orientada a objetos y sus propiedades más importantes: abstracción, encapsulamiento de datos, polimorfismo y herencia junto a las clases y objetos.

El capítulo termina con una breve historia y evolución de los lenguajes de programación C y C++.

REFERENCIAS BIBLIOGRÁFICAS Y LECTURAS RECOMENDADAS

BROOKSHEAR, J. Glenn. *Computer Science*, Eigth ed., Boston (USA): Pearson/Addison-Wesley (2005).

Uno de los mejores libros para aprender la ciencia de la computación a nivel de introducción y medio. Trata todos los temas relacionados con el mundo de la programación con una visión académica y científica notable. Desde el estudio de algoritmos hasta lenguajes de programación, ingeniería de software, bases de datos y una excelente introducción a la inteligencia artificial y teoría de la computación.

British Standards Institute. *C++ estándar.* Madrid: Anaya. Prólogo de Bjarne Stroustrup inventor de C++ (2005).

Esta obra es el Estándar Internacional del lenguaje de programación C++. Esta versión traducida por la editorial Anaya del estándar original en inglés, recoge las especificaciones completas del C++ estándar, oficialmente conocido como BS ISO/IEC 14882:1998 con las actualizaciones (Corrigendum Técnico 1) adoptadas entre 1997 y 2001. Obra de referencia indispensable para el trabajo profesional en C++. Las 1.006 páginas de esta enciclopedia recogen las especificaciones oficiales. Es un libro de consulta para resolver cualquier duda de aplicación de sintaxis, especificaciones, bibliotecas de funciones, de clases, etc.

JOYANES AGUILAR, Luis. *Fundamentos de programación. Algoritmos, estructuras de datos y objetos*, 3.ª ed., Madrid: McGraw-Hill (2003).

Libro de referencia para el aprendizaje de la programación con un lenguaje algorítmico. Libro complementario de esta obra y que ha cumplido ya quince años desde la publicación de su primera edición

JOYANES AGUILAR, Luis. *Programación orientada a objetos*, 2.ª ed., Madrid: McGraw-Hill (2003).

Libro de referencia para el aprendizaje de la programación orientada a objetos y que se escribió en la segunda mitad de la década de los noventa cuando UML comenzaba a emerger. Ahora está en fase de revisión y mejora, esperando publicarse la tercera edición en el segundo semestre de 2006.

JOYANES, Luis y SÁNCHEZ, Lucas. *Programación en C++*. Colección Schaum. Madrid: McGraw-Hill, 2006.

Libro complementario de esta obra y con un carácter eminentemente teórico-práctico. Aunque se puede leer y estudiar de modo independiente, también se ha escrito pensando en completar de modo práctico toda la teoría necesaria para iniciarse en algoritmos y estructura de datos, así como en programación orientada a objetos. Contiene un gran número de ejemplos, ejercicios y problemas resueltos.

LAFORE, Robert. *Object-Oriented Programming in C++*, fourth ed., Indianapolis (Indiana): Sams (2002).

Uno de los mejores libros escritos sobre programación orientada a objetos y C++. Su autor reedita cada cierto tiempo el libro, lo cual permite que el lector esté actualizado en todo momento. Esta edición contiene 1.040 páginas y es uno de los libros más completos que se encuentran en el mercado.

PRATA, Stephen. *C++ Primer Plus*, fifth ed., Indianapolis (Indiana): Sams (2005).

Otro gran libro para aprender a programar C++ y métodos de orientación a objetos. Es también un libro enciclopédico (1.075 páginas) muy completo con profusión de ejemplos y ejercicios complejos resueltos.

PRIETO ESPINOSA, Alberto y PRIETO CAMPOS, Beatriz. *Conceptos de Informática*. Colección Schaum. Madrid: McGraw-Hill (2005).

Excelente libro teórico-práctico para conocer todos los fundamentos de la informática y de gran utilidad para el aprendizaje y dominio de la programación.

PRIETO, A., LLORIS, A. y TORRES, *Introducción a la informática*, 3.ª ed., Madrid: McGraw-Hill (2005).

Uno de los mejores libros teóricos para conocer todos los fundamentos de informática necesarios para iniciarse en carreras de ingeniería y ciencias. Lectura imprescindible y consulta obligada, para profundizar en técnicas de programación tanto en el campo del *hardware* como del *software*.

Capítulo 2

El lenguaje C++. Elementos básicos

Contenido

INTRODUCCIÓN

Una vez que se le ha enseñado a crear sus propios programas, vamos a analizar los fundamentos del lenguaje de programación C++. Este capítulo comienza con un repaso de los conceptos teóricos y prácticos relativos a la estructura de un programa enunciados en el capítulo anterior, dada su gran importancia en el desarrollo de aplicaciones, incluyendo además los siguientes temas:

- creación de un programa;
- elementos básicos que componen un programa;
- tipos de datos en C++ y cómo se declaran;
- concepto de constantes y su declaración;
- concepto y declaración de variables;
- tiempo de vida o duración de variables;
- operaciones básicas de entrada/salida.

CONCEPTOS CLAVE

- Archivo de cabecera.
- `char`.
- cin.
- Código ejecutable.
- Código fuente.
- Código objeto.
- Comentarios.
- Constantes.
- `cout`.
- Directiva `#include`.
- `float/double`.
- Flujos.
- Función `main()`.
- `Identificador`.
- `int`.
- Preprocesador.
- Variables.

2.1. CONSTRUCCIÓN DE UN PROGRAMA EN C++

Los programas C++ están escritos en un lenguaje de programación de alto nivel, utilizando letras, números y otros símbolos que se encuentran localizados en el teclado de su computadora, bien directamente pulsando teclas, bien por combinación de teclas u otros artificios de escritura. Las computadoras, como ya conoce el lector, realmente entienden y ejecutan un lenguaje de bajo nivel denominado lenguaje o *código máquina* (un conjunto de números binarios). Por consiguiente, para que la máquina entienda el programa fuente, se requiere de un proceso de traducción del código fuente al código máquina o binario:

```
1010   1111
0011   0111
0111   0110
...
```

Por consiguiente, se requiere que antes de que un programa se pueda utilizar, se debe someter a diferentes transformaciones en un proceso denominado *traducción*.

Los programas comienzan como una idea en la cabeza del programador que termina convirtiéndose en un programa escrito en un lenguaje de programación tal como C++ o Java. Desarrollan su idea escribiendo el *algoritmo* que resuelve el problema, bien en un lenguaje natural, como el *pseudocódigo*, o bien directamente en lenguaje C++ (no recomendable «casi nunca» y menos en la fase de aprendizaje del programador; en general, casi siempre será recomendable la escritura completa o un esbozo del algoritmo previo al análisis del problema).

Una vez que se ha escrito el código del programa debe proceder a su ejecución. ¿Cómo ejecutar entonces un programa? Las etapas prácticas dependerán del compilador o del entorno de programación que utilice; pero, en cualquier forma, serán muy similares a los siguientes:

1. *Utilizar un editor de texto* para escribir el programa fuente y guardarlo en un archivo llamado *archivo fuente* o *código fuente* (en C++ tiene un nombre y una extensión: `.cpp`).
2. *Compilar el código fuente*. El compilador traduce el código fuente al lenguaje interno de la máquina y lo convierte en un archivo denominado *código objeto*. En C++, el archivo resultante tiene el mismo nombre que el fuente y extensión `.obj` o bien `.o`.
3. Si la etapa anterior tiene éxito, el código objeto se enlaza con las bibliotecas de C++ mediante el enlazador (*linker*). Una biblioteca de C++ contiene el código objeto (máquina) de una colección de funciones (rutinas de las computadoras), que realizan tareas tales como visualizar información en pantalla, calcular la raíz cuadrada de un número, etc. El enlazado o montaje de los códigos objeto se combina también con el código de arranque estándar para producir una versión ejecutable de su programa. El archivo que contiene el producto final de la ejecución del programa se denomina *archivo programa* o *código ejecutable* (nombre igual que el fuente y extensión `.exe`).

La Figura 2.1 resume las etapas de programación para transformar un programa escrito en un lenguaje de alto nivel, como C++, en un programa ejecutable.

Afortunadamente, las etapas de compilación y enlace se realizan automáticamente por el compilador. Algunos sistemas de programación ayuda al desarrollador y proporcionan un Entorno de Desarrollo Integrado, **EDI** (*Integrated Development Environment*, **IDE**) que contiene un editor, un compilador, un enlazador, un gestor de proyectos, un depurador y otras herramientas integradas en un paquete de software. Borland, Microsoft y las herramientas de software libre bajo Linux, soportan **EDI** para el desarrollo de programas en C++.

Los programas contenidos en el libro son genéricos y debe ejecutarlos en su sistema, recomendándole la compatibilidad con el estándar **ANSI/ISO C++**.

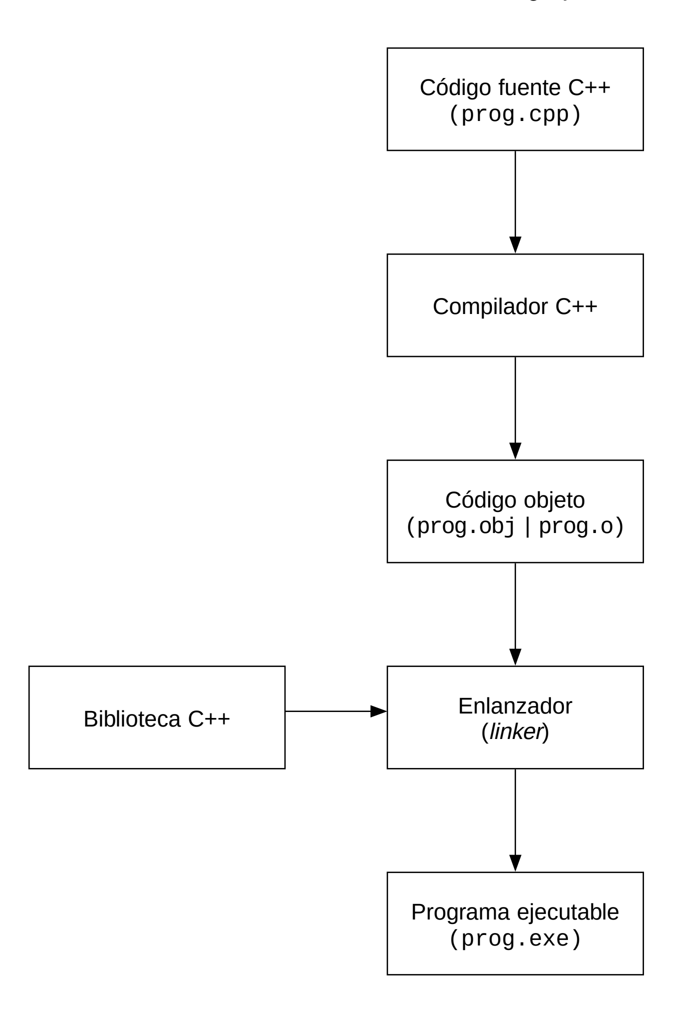


Figura 2.1. Etapas en la ejecución de un programa.

2.1.1. Tipos de compiladores

Existen dos tipos fundamentales de compiladores: *compilador de línea de órdenes* y el *compilador* incluido *en un* **E***ntorno* **I***ntegrado* de **D***esarrollo* —**EID**— (hoy día el más utilizado).

El primer tipo de compilador es el tradicional, *compilador de línea de órdenes* o *compilador activo*. Este tipo de compilador funciona desde la línea de órdenes del sistema operativo, mediante un editor de textos. Una vez terminada la edición del código fuente se ejecuta o teclea una orden y el compilador convierte su código fuente en un programa ejecutable. El segundo tipo de compilador es el ya citado que viene dentro de un Entorno Integrado de Desarrollo.

La mayoría de los sistemas Unix y Linux utilizan compiladores de líneas de órdenes, aunque ya es muy usual la existencia de EID para estos sistemas. Casi todos los compiladores que corren bajo Windows suelen venir con EID incorporados; éstos son los casos de Microsoft Visual C++, Borland C++, Watcom C++, etcétera. Otras implementaciones tales como AT&T C++ o GNU C++ bajo Unix y Linux y las versiones gratuitas de los compiladores de Borland, suelen requerir editores de textos externos y las órdenes de compilación y ejecución se realizan desde la línea de órdenes.

2.1.2. Edición de un programa

En un sistema Unix se pueden utilizar los tradicionales editores `vi`, `ed` o `emacs`. En un sistema DOS se pueden utilizar **`edlin`** o **`edit`** o cualquier otro editor disponible. Se puede utilizar un editor o procesador de textos, tal como Microsoft Word o Notepad; en este caso debe tener la precaución de guardar el archivo en formato texto (ASCII) en lugar de en el formato estándar del procesador de texto, tal como `.doc` en Word, ya que en este caso se producirían errores de compilación y no se podría traducir el programa fuente.

Regla para nombrar un archivo fuente

Al nombrar un archivo fuente se debe utilizar el sufijo adecuado al compilador C++ que esté utilizando. De este modo se identifica el archivo como código fuente C++ y se le indica también al compilador para que lo reconozca. El sufijo consta de un punto seguido por un carácter o grupo de caracteres y se denomina *extensión*.

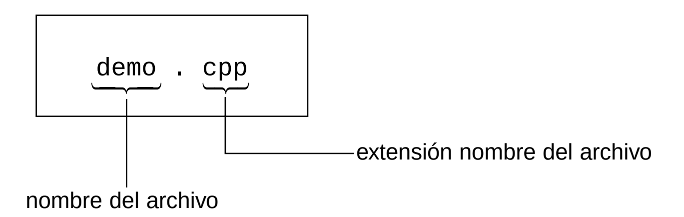


Figura 2.2. Nombre de un archivo en código fuente.

La Tabla 2.1 recoge las extensiones más usuales de los archivos fuente en compiladores C++ comerciales o de libre distribución bajo Linux, que suelen tener por extensión `.cpp`, aunque también existen extensiones `.cc`, `.C` o `.Cxx`, entre otras, aunque todas tienen en común comenzar con `.c`, debido a su origen en el compilador de C. De igual modo los archivos de cabecera o no tienen extensión, caso de los compiladores **ANSI/ISO C++**, o terminan en `.h`, la mayoría; aunque también se encuentran terminados en `.hh`, `.H` o `.hxx`.

TABLA 2.1. Extensiones de códigos fuente

Versión C++	Extensión código fuente*
Microsoft Visual C++	cpp, cc, cxx
Borland C++	cpp
GNU C++	C, cc, cpp, c++, cxx
Unix	C, cc, cxx, c
Watcom	cpp
Dev C++	cpp

* Algunos sistemas son sensibles a las mayúsculas, de modo que en estos casos `Demo.cpp` es un archivo distinto de `demo.cpp`.

Compiladores comerciales y de libre distribución

- **CC** Unix CC Compiler (Unix genérico)
- **g++** Free Software Foundation (gratuito)
- Borland C++ (gratuito)
- Microsoft Visual C++ .NET
- Dev C++
- GNU C++

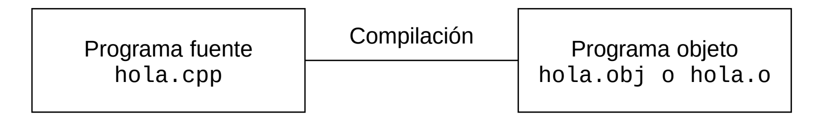
2.1.3. Caso práctico de compilación

1. Crear un programa como archivo de texto con su editor de texto favorito en su EDI (programa fuente, compuesto por una secuencia de caracteres ASCII) y darle el nombre `hola.cpp`.

```
#include <iostream>

int main()
{
   std :: cout << "¡Hola mundo\n";
   return (0)
}
```

2. Compilar el programa con el compilador C++ de GNU, **g++** de Free Software Foundation. Normalmente se ejecuta con la orden C++ o también `g++`.



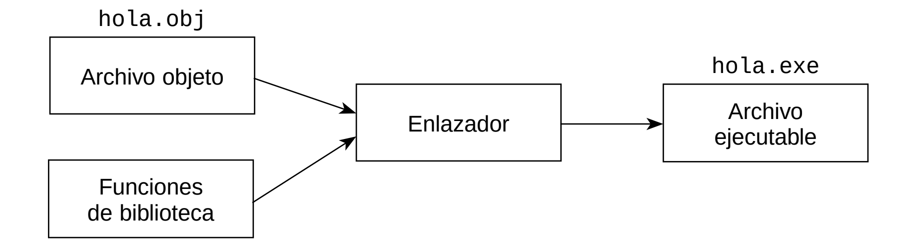
Utilice la línea de órdenes:

```
% g++ -g -Wall -o hola hola.cpp
```

- más simple

```
% C++ -o hola.cpp
```

- *Opción* `-g` para permitir la depuración.
- Interruptor `-wall` activa todos los mensajes de advertencia (*warnings*).
- Interruptor `-o hola` indica al compilador que el programa final se llamará `hola`.
- Programa fuente se llama `hola.cpp`.
- Se crea un archivo ejecutable denominado `hola` (en Unix no suele añadirse la extensión `.exe`) cuando se termina la compilación del programa



3. Ejecutar el programa.
   Se ejecuta el programa desde la línea de órdenes de Linux con la orden

   ```
   hola
   ```

   y se visualiza el mensaje siguiente en la pantalla

   ```
   Hola mundo
   ```

2.1.4. Puesta a punto de un programa en C++

El proceso de creación completo de un programa (*puesta a punto*) ejecutable que realice la tarea prevista y que resuelva un problema determinado entraña las siguientes etapas prácticas:

1. Escritura del código fuente (en papel...).
2. Edición del código fuente (guardar con un nombre; por ejemplo, `demo.cpp`).
3. Compilación del código fuente (`demo.obj`).
4. Detección y reparación de errores de compilación y enlace, ejecución y prueba del programa.

La Figura 2.3 muestra el proceso completo de puesta a punto de un programa.

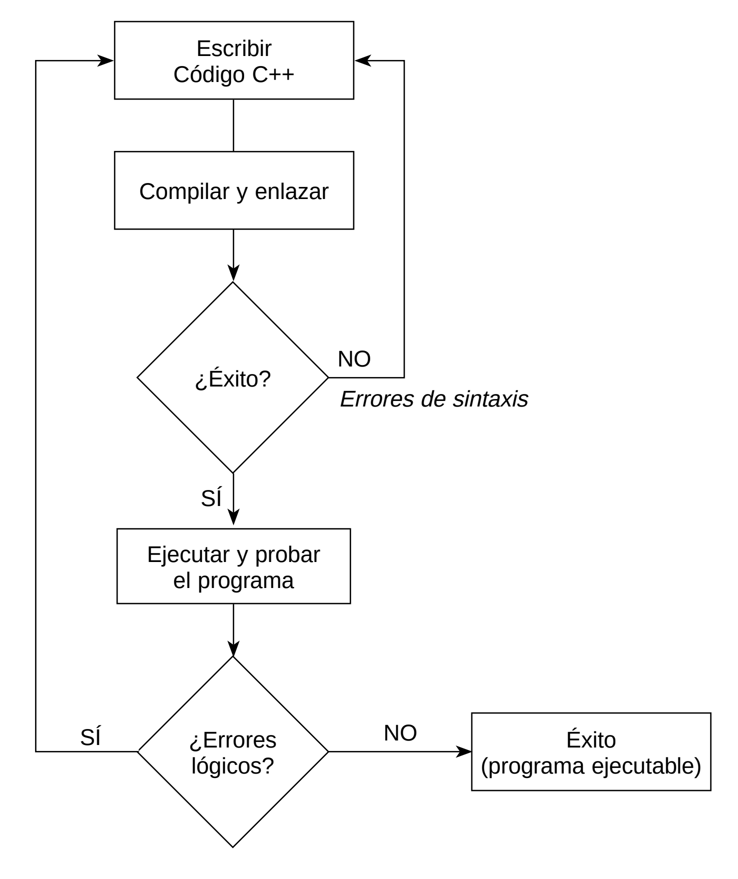


Figura 2.3. Puesta a punto de un programa C++.

Método práctico

Todas las etapas anteriores se resumen de modo práctico en:

1. Introducir el código fuente (las sentencias del programa) con un editor de textos independiente o con el editor incorporado al EID.
2. Compilar el programa (se realiza automáticamente la compilación y el enlace).
3. Ejecutar y probar (*depuración* del programa, *debugging*).

Si el código fuente de su programa no contiene errores cuando se compila el programa se termina con éxito el mismo. Normalmente, el compilador C++ determina si su programa es sintácticamente correcto; si no es así, el compilador le informará de la línea exacta en que ha ocurrido el error.

Sin embargo, puede no haber errores y el programa cuando se ejecuta no se comporta correctamente en alguno o en todos los casos. Después que un programa se ha compilado con éxito, se necesita ejecutar el programa varias veces para verificar que hace exactamente aquello que debe hacer y para lo cual fue diseñado. En el caso de programas comerciales o profesionales que van a ser utilizados por otras personas se requiere mayor exigencia y se debe probar el programa varias veces hasta que se tenga, *casi*, la seguridad de que dicho programa funciona correctamente. Los errrores que se encuentran en la fase de pruebas se denominan *errores de ejecución* o *de lógica* del programa. Con estos errores el compilador dice que el programa ha utilizado la sintaxis correcta, pero por alguna razón el programa no hace lo que debiera hacer, debido a errores lógicos. El proceso de prueba (*testing*) de un programa hasta que funciona correctamente y la detección y reparación de la causa del problema se denomina *depuración* (*debugging*).

Si el programa *corre* (se ejecuta) correctamente, la prueba se ha terminado. Sin embargo, si existen errores de lógica del programa, se necesita determinar la fuente de errores, echar marcha atrás, volver a revisar el código fuente y en muchos casos reconstruir el programa. Si los programas son simples, tal vez con una cantidad moderada de pruebas y reparaciones de los errores, se termina el poblema. Sin embargo, si los programas son complejos, se puede requerir gran cantidad de tiempo, análisis, ejecuciones y pruebas. En estos casos deberá tener paciencia y seguir un método sistemático para la puesta a punto final de su programa.

En la página web del libro puede consultar el proceso completo de edición, compilación y ejecución de programas C++ con sistemas Windows, Linux y Unix tanto con líneas de órdenes como con Entornos Integrados de Desarrollo.

Aclaración

Este libro enseña a programar en ANSI/ISO C++ (C++ estándar). No obstante, a lo largo del mismo se suelen dar reglas y consejos para el caso de que el lector utilice un compilador C++ más antiguo.

El código fuente utilizado en esta obra se debe poder ejecutar en cualquier compilador. Por esta razón no se hace referencia a ventanas, gráficos, cuadros de diálogo y otras herramientas que dependen fundamentalmente del sistema operativo y/o del entorno de desarrollo que se utilice.

2.2. INICIACIÓN A C++: ESTRUCTURA GENERAL DE UN PROGRAMA

Un programa en C++ se compone de una o más funciones. Una de las funciones debe ser obligatoriamente `main`. Una función en C++ es un grupo de instrucciones que realizan una o más acciones. Asimismo, un programa contendrá una serie de directivas `#include` que permitirán incluir en el mismo archivos de cabecera que a su vez constarán de funciones y datos predefinidos en ellos.

De un modo más explícito, un programa C++ puede incluir:

- directivas de preprocesador con `#include`, `using...`
- declaraciones globales;

- la función `main()`;
- funciones definidas por el usuario;
- comentarios del programa (utilizados en su totalidad):
- sentencias.

La estructura típica completa de un programa C++ se muestra en la Figura 2.4. Un ejemplo de un programa sencillo en C++.

```
//Listado DEMO_UNO.CPP. Programa de saludo

#include <iostream>
using namespace std;
// El programa imprime "Bienvenido a la programación C++"
int main()
{
   cout << "Bienvenido a la programación C++\n";
  return 0;
}
```

```
#include <iostream>        ← archivo de cabeza iostream.h
#using namespace std;      ← compilador ANSI/ISO C++
int main()                 ← cabeza de función
{                          ← nombre de la función
     ...                   ← sentencias
}
```

```
#include   Directivas del preprocesador
#define    Macros del procesador
```

Declaraciones globales
- funciones
- variables
- prototipos de funciones

Función principal `main`

```
main ()
{
  declaraciones locales
  sentencias
}
```

Definiciones de otras funciones

```
func1 (...)
{
  ...
}
func2 (...)
{
  ...
}
  ...
```

Figura 2.4. Estructura típica de un programa C++.

La directiva `#include` de la primera línea es necesaria para que el programa tenga salida. Se refiere a un archivo externo denominado `iostream.h` en el que se proporciona la información relativa al objeto `cout`. Obsérvese que los ángulos < y > no son parte del nombre del archivo; se utilizan para indicar que el archivo es un archivo de la biblioteca estándar C++.

La segunda línea es la directiva `using` que permite incluir el espacio de nombre `(namespace) std` y actuar con el flujo de salida `cout`.

La tercera línea es un *comentario*, identificado por dobles barras inclinadas (`//`). Los comentarios se incluyen en programas que proporcionan explicaciones a los lectores de los mismos. Son ignorados por el compilador.

La cuarta línea contiene la cabecera de la función `main()`, obligatoria en cada programa C++. Indica el comienzo del programa y requiere los paréntesis `()` a continuación de `main()`.

La quinta y octava línea contienen sólo las llaves `{` y `}` que encierran el cuerpo de la función `main()` y son necesarias en todos los programas C++.

La sexta línea contiene la sentencia

```
cout << "Bienvenido a la programación C++\n";
```

que indica al sistema que envía el mensaje `"Bienvenido a la programación en C++\n"` al objeto `cout`. Este objeto es el *flujo estándar de salida* que normalmente representa la pantalla de presentación de la computadora (el nombre `cout` es una abreviatura de *«console output»* —salida a consolas). La salida será

```
Bienvenido a la programación C++
```

El símbolo `'\n'` es el símbolo de *nueva línea*. Poniendo este símbolo al final de la cadena entre comillas, indica al sistema que comience una nueva línea después de imprimir los caracteres precedentes, terminando, por consiguiente, la línea actual.

La séptima línea contiene la sentencia `return 0`. Esta sentencia termina la ejecución del programa y devuelve el control al sistema operativo de la computadora. El número 0 se utiliza para señalar que el programa ha terminado correctamente (*con éxito*).

La sentencia de salida de la sexta línea incluye diversos símbolos típicos de C++. El símbolo << se denomina *operador de salida* u *operador de inserción*. Inserta el mensaje en el flujo de salida. El símbolo `\n` incluido al final del mensaje significa *carácter de nueva línea*. Siempre que aparece en un mensaje de salida, se termina la línea actual de salida y comienza una nueva línea.

Obsérvese el punto y coma (`;`) al final de la sexta y séptima línea. C++ requiere que cada sentencia termine con un punto y coma. No es necesario que esté al final de una línea. Se pueden poner varias sentencias en la misma línea y se puede hacer que una sentencia se extienda sobre varias líneas.

Advertencia

- El programa más `corto` de C++ es el programa vacío que no hace nada:
  ```
  int main()
  {}
  ```
- La sentencia `return 0;` no es obligatoria en la mayoría de los compiladores, aunque algunos emiten un mensaje de advertencia si se omite.

2.2.1. El preprocesador de C++ y las directivas

C++, igual que C, utiliza un **preprocesador**. El ***preprocesador*** es un programa que procesa un archivo fuente antes de que tenga lugar la compilación principal. En la práctica, el preprocesador prepara el có-

digo fuente para que el compilador pueda realizar correctamente su compilación. Permite incluir en el código otros archivos (denominados *cabecera*), definir macros, eliminar los comentarios, etc. No se requiere hacer nada especial para invocar a este procesador, ya que funciona automáticamente cuando se cumpla el programa.

El preprocesador en un programa C o C++ se puede considerar como un editor inteligente de texto que se compone de *directivas* (**`instrucciones`** al compilador antes de que se compile el programa principal). Los programas en C++ suelen comenzar con directivas. Las directivas más usuales son `#include`, `#define` y `using` (en la última versión de C++, el estándar ANSI/ISO C++). El preprocesador se llama automáticamente cuando se compila a su programa.

Directa `#include`

La directiva `#include` —una de las muchas directivas del preprocesador— indica al procesador que inserte otro archivo en su archivo fuente. En efecto, la directiva `#include` es reemplazada por el contenido del archivo indicado a continuación. En la práctica, usar una directiva `#include` para insertar otro archivo en su archivo fuente es similar a la tarea de «pegar» un bloque de texto en un documento con un procesador de textos. El archivo de texto que se incluye en `#include` y en otras directivas se denomina *archivo de cabecera*.

```
#include <nombre_archivo_cabecera>
#include <iostream> inserta el contenido del archivo iostream
```

El archivo `iostream` es muy importante en los programas C++, ya que facilita la comunicación entre el programa y el mundo exterior. La `io` (*input-output*) de `iostream` se refiere a la *entrada* (*input*), información que llega al programa, y la *salida* (*output*), que es la información enviada fuera del programa.

La directiva `#include` hace que el contenido del archivo `iostream` se envíe junto con el contenido de su archivo al compilador. En esencia, el contenido del archivo `iostream` reemplaza a la línea `#include <iostream>` en el programa. Su archivo original no se altera, pero en la siguiente etapa de compilación se forma un archivo compuesto de su archivo fuente y el archivo `iostream`.

El archivo de cabecera `iostream` incluye las declaraciones para poder utilizar los identificadores `cunt` y `cin` para sacar datos e introducir datos.

Los programas que utilizan `cin` y `cunt` para entrada y salida de datos deben incluir el archivo `iostream` en C++ estándar y `iostream.h` en las versiones antiguas.

Archivos de cabecera

Los archivos tales como `iostream` se denominan *archivos de inclusión* (debido a que se incluyen en otros archivos) o *archivos de cabecera* (ya que se incluyen al principio de un archivo).

Los compiladores de C++ incorporan muchos archivos de cabecera que soportan cada uno diferentes características. Los archivos de cabecera de C utilizaban la extensión `.h` con su nombre. Por ejemplo, el

Tabla 2.2. Notaciones de nombres de archivos de cabecera

Tipo de cabecera	Notación	Ejemplo	Uso en
Estilo C antiguo	`.h`	`math.h`	Programas C y C++
Estilo C++ antiguo	`.h`	`iostream.h`	Programas C++
Estilo nuevo estándar C++	Ninguna extensión	`iostream`	Programas C++, recomendado usar `using namespace std`
Otros archivos de cabeceras nuevo estándar C++	Prefijo `c` ninguna extensión	`cmath`	Programas C++

archivo de cabecera `math.h` soporta diferentes funciones matemáticas. En principio, C++ tenía la misma representación. Así, por ejemplo, el archivo de cabecera `iostream.h` contenía facilidades para entrada y salida. En los compiladores nuevos de C++ se ha cambiado la notación y ya los archivos de cabecera no tienen extensión. La extensión `.h` se reserva para los compiladores de C o las versiones de C++ anteriores al estándar ANSI/ISO. Los nuevos archivos de C++ comienzan con la letra `c` y se ha suprimido la extensión `.h`.

La directiva `#using`

Si se utiliza el archivo `iostream` del estándar ANSI/ISO C++ en lugar de `iostream.h`, propio de las versiones antiguas de C++, deberá utilizar la directiva `using` de *espacio de nombres* para permitir que las definiciones de `iostream` estén disponibles en su programa.

Esta necesidad viene de que un programa C++ se puede devidir en diferentes espacios de renombres (*namespaces*). Un **espacio de nombres**, como se verá posteriormente, es una parte del programa en la cual ciertos nombres son reconocidos y fuera de ese espacio son desconocidos.

La directiva `using namespace std;`

La directiva indica que todas las sentencias del programa que vienen a continuación están dentro del espacio de nombres `std`. Diversos componentes de programa, tales como `cout`, están declarados dentro de este espacio de nombres.

Ejemplo

Uno de la directiva `using` *y reconocimiento de* `cout`

```
#include <iostream>
#using namespace std;

int main()
{
   cout << "Hola mundo C++\n";
   return 0;
}
```

Si no se utiliza la directiva `using` se necesitará preceder el nombre `std` a muchos elementos del programa.

Ejemplo

Programa sin directiva `using namespace std;`

```
#include <iostream>

int main()
{
   std::cout << "Hola mundo, C++\n";
   return 0;
}
```

Reglas prácticas

1. Si desea que su programa utilice las facilidades de entrada y salida (`cin` y `cout`) del estándar C++, debe proporcionar estas dos líneas al principio

```
#include <iostream>
using namespace std;
```

2. Si su compilador le presenta problemas al compilar y le avisa que no encuentra el archivo `iostream`, es que está utilizando un compilador antiguo no compatible con el estándar, y entonces debe sustituir las dos líneas anteriores por

```
#include <iostream.h>   //compatible con compiladores antiguos
```

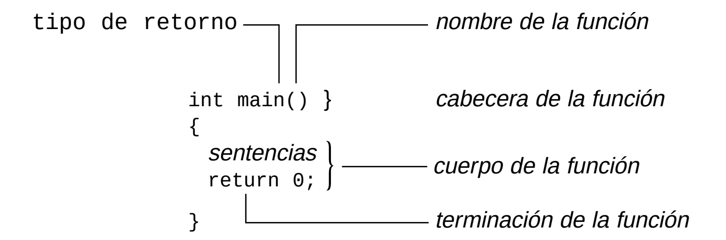
3. Para evitar añadir numerosas veces `std::...` en los programas, utilizaremos casi siempre a lo largo del texto la directiva `using namespace std;`

> ***Nota***
>
> En todos los programas de la primera edición de esta obra se incluía `#include <iostream.h>`, que reconoce automáticamente `cin` y `cout`.

2.2.1. Función `main()`

Cada programa de C++ contiene una función `main()` que es la función principal y que es el primer punto de entrada al programa. Cuando se ejecuta el programa, se invoca, en primer lugar, a la función `main()`. Cualesquiera otra función se llama directa e indirectamente desde `main`.

La función `main()` tiene la siguiente estructura: *nombre, cabecera, cuerpo, definición* y *terminación de la función* (Figura 2.5). El cuerpo de la función `main()` es un conjunto de sentencias que se ejecutan cuando se ejecuta `main` y termina con una última sentencia.



Nota: Las sentencias terminan con un punto y coma
`return 0;` termina la función `main`

Figura 2.5. Función `main`.

En C++, una *sentencia* representa una instrucción completa a la computadora. Cada sentencia se debe terminar con un punto y coma, por lo que debe recordar no omitir los caracteres punto y coma cuando escriba sus programas.

En general, una función C++ se *activa* o *llama* por otra función y la cabecera de la función describe la interfaz entre una función y la función que la llama (función llamadora o invocadora). La parte que precede al nombre de la función se denomina *tipo de retorno* de la función, que describe el flujo de in-

formación desde una función a la función invocadora. La parte dentro de los paréntesis, a continuación del nombre, se denomina *lista de argumentos* o *lista de parámetros*, y describe el flujo de información de la función, desde la función llamadora a la función llamada.

En el caso de `main()` no se consideran argumentos, ya que `main`, normalmente, no se llama desde otras partes de su programa. En general, `main` es llamada por el código inicial que el compilador añade a su programa para conectar el programa y el sistema operativo (Unix, Linux, Windows XP, Windows Vista, etc.), y por ello actúa como la interfaz entre `main` y dicho sistema operativo.

Una función C++ invocada por otra función puede devolver un valor a la función que le llama. Este valor se denomina *valor de retorno*. En el caso de `main()` puede devolver un valor de tipo entero cuando se indica con la palabra reservada `int`.

El formato más aceptado de `main()` es

```
int main()
main()
```

cuyo significado no tiene parámetros y devuelve un valor entero (`int`).

Algunos compiladores permiten omitir `int` o sustituirla por `void` indicando que la función no devuelve ningún valor. El programa termina cuando se ejecuta la sentencia:

```
return 0;
```

que devuelve cero como valor de la función. Aunque el estándar ANSI/ISO C++ no exige esta sentencia, pero como muchos compiladores requieren su uso, recomendamos manternerla dentro de la función `main()`. La línea `return(0)` indica al sistema operativo que el programa terminó normalmente con éxito (estado 0). Un estado distinto de cero indica un error; normalmente 1 se utiliza para los errores más simples, tales como sintaxis incorrecta o archivo no encontrado.

Los paréntesis vacíos de `main()` significan la inexistencia de argumentos. En general, una función puede pasar información a otra función cuando se la invoca y en este caso dicha información está contenida entre paréntesis.

Reglas de estilo

C	Cabecera de la función	`main()`
C++	Cabecera de la función	`int main()`
	alternativa	`int main(void)`

Recuerde ANSI/ISO C++

- Si el compilador alcanza el final de `main()` sin encontrar una sentencia `return`, asume por defecto que termina con la sentencia:

  ```
  return (0);
  ```

- El retorno implícito sólo existe en `main()` y no en otras funciones.
- Algunos programadores utilizan la siguiente cabecera y omiten la sentencia `return`

  ```
  void main()
  ```

  que implica la misma acción de `return`, ya que un tipo de retorno `void` significa que no devuelve un valor.

2.2.2. Declaraciones globales

Las *declaraciones globales* indican al compilador que las funciones definidas por el usuario o variables así declaradas son comunes a todas las funciones de su programa. Las declaraciones globales se sitúan antes de la función `main()`.

Si se declara global una variable `Grado_clase` del tipo

```
int Grado_clase;
```

cualquier función de su programa, incluyendo `main()`, puede acceder a la variable `Grado_clase`.

La zona de declaraciones globales de un programa puede incluir declaraciones de variables además de declaraciones de función.

Las declaraciones de función se denominan *prototipos.*

```
int media(int a, int b);
```

El siguiente programa es una estructura modelo que incluye declaraciones globales.

```
// Programa demo.cpp
#include <iostream.h>

// Definir macros
#define MICONST1 0.50

#define MICONS2 0.75

//Declaraciones globales
int Calificaciones ;
main()
{
   // ...
}
```

2.2.3. Funciones definidas por el usuario

Un programa C++ es una colección de funciones. Todos los programas se construyen a partir de una o más funciones que se integran para crear una aplicación. Todas las funciones contienen una o más sentencias C++ y se crean generalmente para realizar una única tarea, tales como imprimir la pantalla, escribir un archivo o cambiar el color de la pantalla. Se pueden declarar y ejecutar un número de funciones casi ilimitado en un programa C++.

Las funciones definidas por el usuario se invocan por su nombre y los parámetros opcionales que puedan tener. Después de que la función se ejecuta, el código asociado con la función se ejecuta y, a continuación, se retorna a la función llamadora.

Todas las funciones tienen nombre y una lista de valores que reciben. Se puede asignar cualquier nombre a su función, pero normalmente se procura que dicho nombre describa el propósito de la función.

En C++, las funciones requieren una *declaración* o *prototipo* en el programa:

```
void func demo();
```

Una *declaración de función* indica al compilador el nombre de la función que se está invocando en el programa. Si la función no se define, el compilador informa de un error. La palabra reservada `void` significa que la función no devuelve un valor.

```
void contararriba(int valor);
```

La definición de una función es la estructura de la misma

`tipo_retorno nombre_función(lista_de_parámetros)`	*principio de la función*
`{`	
`   sentencias`	*cuerpo de la función*
`   return;//opcional`	*retorno de la función*
`}`	*fin de la función*

`tipo_retorno`	*tipo de valor, o `void`, devuelto por la función*
`nombre_función`	*nombre de la función*
`lista_de_parámetros`	*Lista de parámetros, o `void`, pasados a la función. Se conoce también como argumentos de la función o argumentos formales.*

C++ proporciona también funciones predefinidas que se denominan *funciones de biblioteca*. Las funciones de biblioteca son funciones listas para ejecutar que vienen con el lenguaje C++. Requieren la inclusión del archivo de cabecera estándar, tal como `STDIO.H`, `MATH.H`, etc. Existen centenares de funciones definidas en diversos archivos de cabecera.

```
// ejemplo funciones definidas por el usuario

#include <iostream>
using namespace std;

int visualizar();
int main()
{
  visualizar();
  return 0;
}

void visualizar()
{
  cout << "Hola mundo guay\n";
}
```

Los programas C++ constan de un conjunto de funciones que normalmente están controladas por la función `main()`.

```
main()
{
  //...
}

obtenerdatos()
{
  //...
}

alfabetizar()
{
  //...
}
//...
```

2.2.4. Comentarios

Un *comentario* es cualquier información que se añade a su archivo fuente para proporcionar información de cualquier tipo. El compilador ignora los comentarios, no realiza ninguna tarea concreta. El uso de comentarios es totalmente opcional, aunque dicho uso es muy recomendable.

Generalmente, se considera buena práctica de programación comentar su archivo fuente tanto como sea posible, al objeto de que usted mismo y otros programadores puedan leer fácilmente el programa con el paso del tiempo. Es buena práctica de programación comentar su programa en la parte superior de cada archivo fuente. La información que se suele incluir es el nombre del archivo, el nombre del programador, una breve descripción, la fecha en que se creó la versión y la información de la revisión.

En C++ los comentarios de un programa se pueden introducir de dos formas:

- *estilo C estándar;*
- *estilo C++.*

Si sólo se está utilizando el compilador de C++ se recomienda el uso del estilo C++, reservando el estilo C para aquellos casos en que se necesita ejecutar su programa bajo otro compilador de C.

Estilo C estándar

Los comentarios en C estándar comienzan con la secuencia `/*` y terminan con la secuencia `*/`. Todo el texto situado entre las dos secuencias es un comentario ignorado por el compilador.

```
/* PRUEBA1.CPP  — Primer programa C++ */
```

Si se necesitan varias líneas de programa se puede hacer lo siguiente:

```
/*
  Programa:        PRUEBA1.CPP
  Programador:     Pepe Mortimer
  Descripción:     Primer programa C++
  Fecha creación:  17 junio 1994
  Revisión:        Ninguna
*/
```

También se pueden situar comentarios de la forma siguiente:

```
cout << "Programa Demo";/* sentencia de salida */
```

Estilo C++

Los comentarios estilo C++ son nuevos. Se define una línea de comentario comenzando con una doble barra inclinada (`//`); todo lo que viene después de la doble barra inclinada es un comentario y el compilador lo ignora. Un comentario comienza con `//` y termina al final de la línea en la cual se encuentra el símbolo.

```
// PRUEBA1.CPP     -- Primer programa C++
```

Si se necesitan varias líneas de comentarios, se puede hacer lo siguiente:

```
//
//Programa:        PRUEBA1.CPP
//Programador:     Pepe Mortimer
```

```
//Descripción:     Primer programa C++
//Fecha creación:  17 junio 1994
//Revisión:        Ninguna
//
```

No se pueden anidar comentarios, lo que implica que no es posible escribir un comentario dentro de otro. Si se anidan comentarios, el compilador produce errores, ya que no puede discernir entre los comentarios.

El comentario puede comenzar en cualquier parte de la línea, incluso después de una sentencia de programa. Por ejemplo,

```
// PRUEBA1.CPP  -- Primer programa C++
#include <iostream>              //archivo de cabecera
using namespace std;

int main()                           //función principal
{
  cout << "Hola mundo guay";     //visualiza en pantalla
  return 0;                      //fin de la función y
                                 //devolución de 0
}
```

Ejemplo 2.1

Supongamos que se ha de imprimir su nombre y dirección muchas veces en su programa C++. El sistema normal es teclear las líneas de texto cuantas veces sea necesario; sin embargo, el método más rápido y eficiente sería escribir el código fuente correspondiente una vez *y, a continuación, grabar un archivo* `midirec.cpp`*, de modo que para incluir el código sólo necesitará incluir en su programa la línea*

```
#include <midirec.cpp>
```

Es decir, teclee las siguientes líneas y grábelas en un archivo denominado `MIDIREC.CPP`

```
// archivo midirec.cpp
cout << "Luis Joyanes Aguilar\n";
cout << "Avda de Andalucía, 48\n";
cout << "Carchelejo, JAEN\n";
cout << "Andalucía, ESPAÑA\n";
```

El programa siguiente:

```
// nombre del archivo demoinc1.cpp
// ilustra el uso de # include

#include <iostream>
using namespace std;

int main()
{
  #include "midirec.cpp"
  return 0;
}
```

equivale a

```
// nombre del archivo demoinc1.cpp
//ilustra el uso de #include

#include <iostream>
using namespace std;

int main()

{
  cout << "Luis Joyanes Aguilar\n";
  cout << "Avda de Andalucía, 48\n";
  cout << "Carchelejo, JAEN\n";
  cout << "Andalucía, ESPAÑA\n";
  return 0;
}
```

Ejemplo 2.2

El siguiente programa copia un mensaje en un array de caracteres y lo imprime en la pantalla ya que `cout` *y* `strcpy()`[1] *(una función de cadena) se utilizan, se necesitan sus archivos de cabecera específicos.*

```
// nombre del archivo demoinc2.cpp
// utiliza dos archivos de cabecera
#include <iostream>
#include <string>
using namespace std;

int main()
{
  char mensaje[20];
  strcpy (mensaje, "hola mundo\n");
  //
Las dos líneas anteriores también se pueden sustituir por
  // char mensaje[20] = "hola mundo\n";
  cout << mensaje;
  return 0;
}
```

> Los archivos de cabecera en C++ tienen normalmente una extensión `.h` o bien `.hpp` y los archivos fuente, la extensión `.cpp`.

2.2.5. Funciones de biblioteca

Muchas de las actividades de C++ se realizan con *funciones de biblioteca*. Estas funciones realizan acceso a archivos, cálculos matemáticos y conversiones de datos, entre otras actividades. Aunque se dedicará un capítulo específico a funciones C++, el uso de las funciones de biblioteca es muy sencillo, y por ello se incluyen aquí para que el lector se vaya familiarizando con ellas.

[1] La función `strcpy()` asigna una secuencia o cadena de caracteres a una variable tipo carácter en el Ejemplo 2.2, a `mensaje`. En los Capítulos 8 y 11 se explicarán en detalle el concepto de cadena y arrays de caracteres.

Hasta este momento ya se han utilizado funciones definidas por el usuario, tales como `main()`; veamos cómo utilizar otras funciones incluidas en otras bibliotecas como `sqrt` de la biblioteca `cmat` (o su equivalente `math` en compiladores antiguos).

Ejemplo 2.3

Utilizar la función de biblioteca `sqrt()` *para calcular la raíz cuadrada de un número introducido por el usuario.*

En primer lugar, para utilizar `sqrt()` en un programa se debe proporcionar el prototipo de la función. En este caso se puede hacer de dos formas:

- Teclear el prototipo de la función (en su código fuente).
- Incluir el archivo de cabecera `cmath` (`math.h` en sistemas C++ antiguos) que tiene el prototipo incluido.

No se debe confundir el prototipo con la definición de la función. El *prototipo* describe la interfaz de la función, que es la información enviada a la función y la información devuelta. La *definición* incluye el código fuente, que realiza la tarea correspondiente.

C++ contiene estas dos características en las funciones de biblioteca. Los archivos de la biblioteca contienen el código compilado de la función, mientras que los archivos compilados contienen los prototipos.

Prototipo de la función `sqrt()`

```
   double sqrt(double);
```

`double`	*la función* `sqrt()` *devuelve un valor de tipo* `double` *(real)*
`(double)`	*la función* `sqrt()` *requiere un argumento de tipo* `double` *(real)*

Ejemplo

```
double n;           //n se declara como tipo double
n = sqrt (16.25);
```

Precaución

El punto y coma del prototipo identifica que es una sentencia; si se omite el punto y coma, el compilador interpreta que es una cabecera de función y espera que siga a continuación el cuerpo de la función que define dicha función.

Ejemplo 2.4

Sintaxis de llamada a la función `sqrt()`

valor devuelto de la función asignado a `n`

argumento, información pasada a la función

```
n = sqrt (16.25);
```

punto y coma, final de la sentencia

paréntesis apertura y cierre

nombre de la función

Código fuente

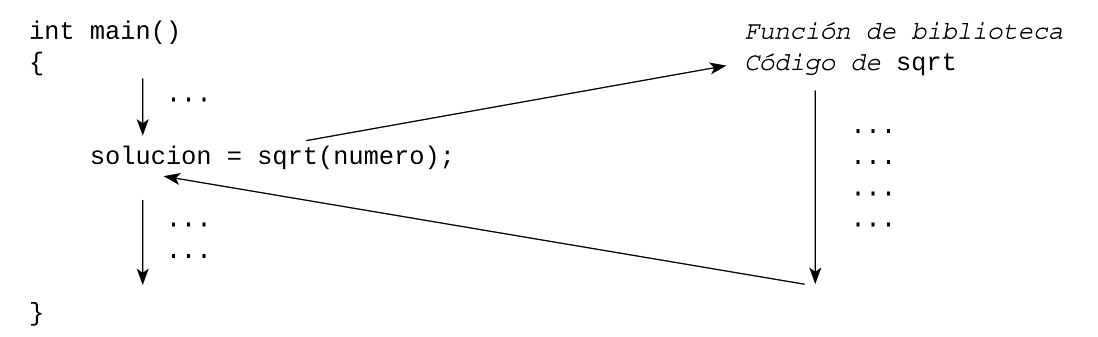
```
//archivo raizcua.cpp
//uso de la función de biblioteca sqrt()
#include <iostream>      //uso de cont, cin, ...
#include <cmath>         //uso de sqrt(), también math.h
using namespace std;

int maint()
{
   double numero, solucion;               //sqrt rquiere argumento real

   cout << "Introduzca un número: ";      //pedir número
   cin >> numero;                         //leer número
   solución = sqrt(numero):               //obtener raíz cuadrada
   cout << "La raíz cuadrada es" << solución << endl;
   return 0;
```

> Si utiliza un compilador antiguo, utilice <`math.h`> en lugar de <`math`>.

Un diagrama de la ejecución del programa `raizcua` es:



2.3. CREACIÓN DE UN PROGRAMA

Una vez construido y editado un programa en C++ como el anterior, se debe ejecutar. ¿Cómo realizar esta tarea? Los pasos a dar dependerán del compilador C++ que utilice. Sin embargo, serán similares a los mostrados en la Figura 2.6. En general, los pasos serían:

- Utilizar un editor de texto para escribir el programa y grabarlo en un archivo. Este archivo constituye el código *fuente* de un programa.
- Compilar el código fuente. Se traduce el código fuente en un *código objeto* (extensión `.obj`) (lenguaje máquina entendible por la computadora). Un *archivo objeto* contiene instrucciones en lenguaje máquina que se pueden ejecutar por una computadora. Los archivos de estándar C++ y los de cabecera definidos por el usuario son incluidos (`#include`) en su código fuente por el preprocesador. Los archivos de cabecera contienen información necesaria para la compilación, como es el caso de `iostream` e `iostream.h` que contiene información de `cout` y de <<.
- Enlazar el código objeto con las *bibliotecas* correspondientes. Una biblioteca C++ contiene código objeto de una colección de rutinas o *funciones* que realizan tareas, como visualizar informaciones en la pantalla o calcular la raíz cuadrada de un número. El enlace del código objeto del programa

con el objeto de las funciones utilizadas y cualquier otro código empleado en el enlace, producirá un código *ejecutable*. Un programa C++ consta de un número diferente de archivos objeto y archivos biblioteca.

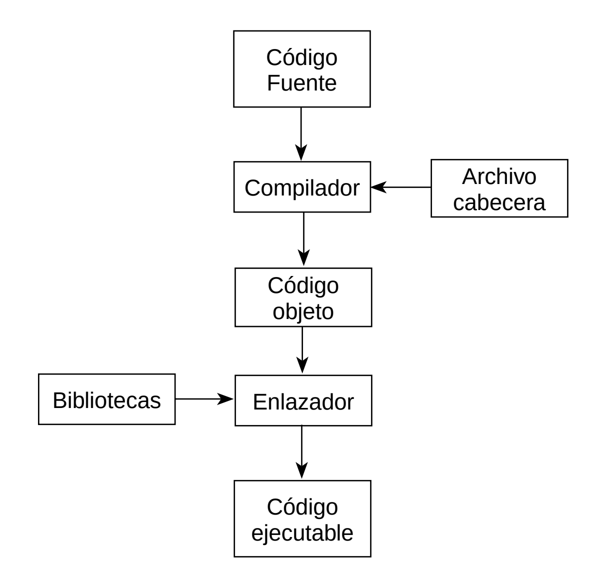


Figura 2.6. Etapas de creación de un programa.

Para crear un programa se utilizan las siguientes etapas:

1. Definir su programa.
2. Definir directivas del preprocesador.
3. Definición de declaraciones globales.
4. Crear `main()`.
5. Crear el cuerpo del programa.
6. Crear sus propias funciones definidas por el usuario.
7. Compilar, enlazar, ejecutar y comprobar su programa.
8. Utilizar comentarios.

2.4. EL PROCESO DE EJECUCIÓN DE UN PROGRAMA EN C++

Un programa de computadora escrito en un lenguaje de programación (por ejemplo, C++) tiene forma de un texto ordinario. Se escribe el programa en una hoja de papel y a este programa se le denomina *programa texto* o *código fuente*. Considérese el ejemplo sencillo y clásico de Bjarne Stroustrup

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hola mundo, C++!" << endl;
  return 0;
}
```

La primera operación en el proceso de ejecución de un programa es introducir las sentencias (instrucciones) del programa en un editor de texto. El editor almacena el texto y debe proporcionarle un nombre tal como `hola.cpp`. Si la ventana del editor le muestra un nombre tal como `noname.cpp`, es

conveniente cambiar dicho nombre (por ejemplo, por `hola.cpp`). A continuación, se debe guardar el texto en disco para su conservación y uso posterior, ya que en caso contrario el editor sólo almacena el texto en memoria central (RAM) y cuando se apague la computadora, o bien ocurra alguna anomalía, se perderá el texto de su programa. Sin embargo, si el texto del programa se almacena en un disquete, en un disco duro, o bien en un CD-ROM, el programa se guardará de modo permanente, incluso después de apagar la computadora y siempre que ésta se vuelva a arrancar.

La Figura 2.7 muestra el método de edición de un programa y la creación del programa en un disco, en un archivo que se denomina *archivo de texto*. Con la ayuda de un editor de texto se puede editar el texto fácilmente, es decir, cambiar, mover, cortar, pegar, borrar texto. Se puede ver, normalmente, una parte del texto en la pantalla y se puede marcar partes del texto a editar con ayuda de un ratón o el teclado. El modo de funcionamiento de un editor de texto y las órdenes de edición asociadas varían de un sistema a otro.

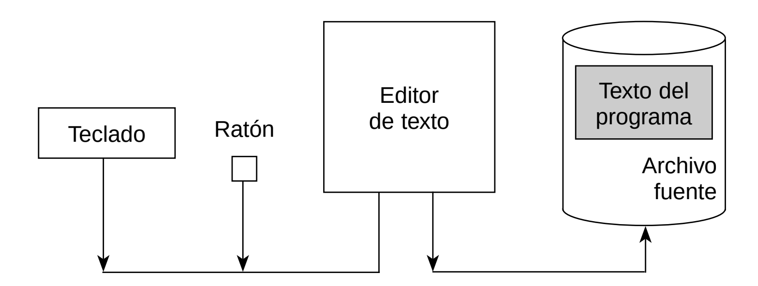


Figura 2.7. Proceso de edición de un archivo fuente.

Una vez editado un programa, se le proporciona un nombre. Se suele dar una extensión al nombre (normalmente `.cpp`, aunque en algunos sistemas puede tener otros sufijos tales como `c` o bien `cc`).

La siguiente etapa es la de *compilación*. En ella se traduce el código fuente escrito en lenguaje C++ a código máquina (entendible por la computadora). El programa que realiza esta traducción se llama *compilador*. Cada compilador se construye para un determinado lenguaje de programación (por ejemplo, C++); un compilador puede ser un programa independiente (como suele ser el caso de sistemas operativos como Linux, Windows, UNIX, etc.) o bien formar parte de un programa entorno integrado de desarrollo (**EID**). Los programas EID (**EDE**, en inglés) contienen todos los recursos que se necesitan para desarrollar y ejecutar un programa, por ejemplo, editores de texto, compiladores, enlazadores, navegadores y depuradores.

Cada lenguaje de programación tiene unas reglas especiales para la construcción de programación que se denomina *sintaxis*. El compilador lee el programa del archivo de texto creado anteriormente y comprueba que el programa sigue las reglas de sintaxis del lenguaje de programación. Cuando se compila su programa, el compilador traduce el código fuente C++ (las sentencias del programa) en un código máquina (*código objeto*). El código objeto consta de instrucciones máquina e información de cómo cargar el programa en memoria antes de su ejecución. Si el compilador encuentra errores, los presentará en la pantalla. Una vez corregidos los errores con ayuda del editor se vuelve a compilar sucesivamente hasta que no se produzcan errores.

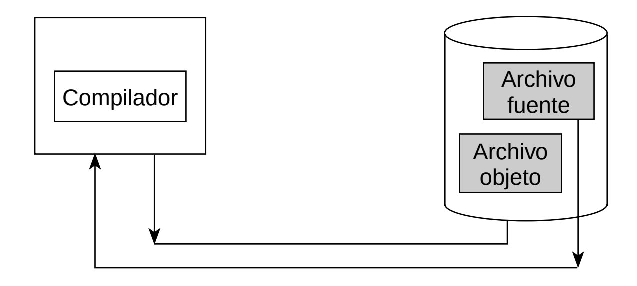


Figura 2.8. Proceso de compilación de un programa.

El código objeto así obtenido se almacena en un archivo independiente, normalmente con extensión `.obj` o bien `.o` . Por ejemplo, el programa `hola` anterior se puede almacenar con el nombre `hola.obj`.

El archivo objeto contiene sólo la traducción del código fuente. Esto no es suficiente para ejecutar realmente el programa. Es necesario incluir los archivos de biblioteca (por ejemplo, en el programa `hola.cpp`, `iostream.h`, `iostream`). Una biblioteca es una colección de código que ha sido programada y traducida y lista para utilizar en su programa.

Normalmente, un programa consta de diferentes unidades o partes de programa que se han compilado independientemente. Por consiguiente, puede haber varios archivos objetos. Un programa especial llamado *enlazador (linker)* toma el archivo objeto y las partes necesarias de la biblioteca `iostrean` y construye un *archivo ejecutable*. Los archivos ejecutables tienen un nombre con la extensión `.exe` (en el ejemplo, `hola.exe)` o simplemente `hola,` según sea su computadora. Este archivo ejecutable contiene todo el código máquinas necesario para ejecutar el programa. Se puede ejecutar el programa escribiendo `hola` en el indicador de órdenes o haciendo clic en el icono del archivo.

Se puede poner ese archivo en un DVD o en un CD-ROM, de modo que esté disponible después de salir del entorno del compilador a cualquier usuario que no tenga un compilador C++ o que puede no conocer lo que hace.

El proceso de ejecución de un programa no suele funcionar la primera vez; es decir, casi siempre hay errores de sintaxis o errores en tiempo de ejecución. El proceso de detectar y corregir errores se denomina *depuración* o *puesta a punto* de un programa.

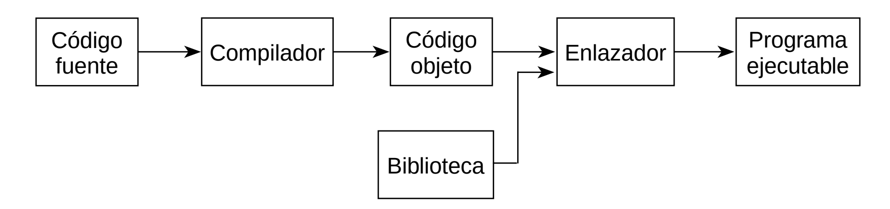


Figura 2.9. Proceso de conversión de código fuente a código ejecutable.

La Figura 2.10 muestra el proceso completo de puesta a punto de un programa.

Se comienza escribiendo el archivo fuente con el compilador. Se compila el archivo fuente y se comprueban mensajes de errores. Se retorna al editor y se fijan los errores de sintaxis. Cuando el compilador tiene éxito, el enlazador construye el archivo ejecutable. Se ejecuta el archivo ejecutable. Si se encuentra un error, se puede activar el depurador para ejecutar sentencia a sentencia. Una vez que se encuentra la causa del error, se vuelve al editor y se corrige. El proceso de compilar, enlazar y ejecutar el programa se repetirá hasta que no se produzcan errores.

Etapas del proceso

- El código fuente (archivo del programa) se crea con la ayuda del editor de texto.
- El compilador traduce el archivo texto en un archivo objeto.
- El enlazador pone juntos a diferentes archivos objetos para poner un archivo ejecutable.
- El sistema operativo pone el archivo ejecutable en la memoria central y se ejecuta el programa.

2.5. DEPURACIÓN DE UN PROGRAMA EN C++

Como se ha comentado anteriormente, rara vez los programas funcionan bien la primera vez que se ejecutan. Los errores que se producen en los programas han de ser detectados, aislados (fijados) y corregidos. El proceso de encontrar errores se denomina **depuración** del programa. La corrección del error es probablemente la etapa más fácil, siendo la detección y aislamiento del error las tareas más difíciles.

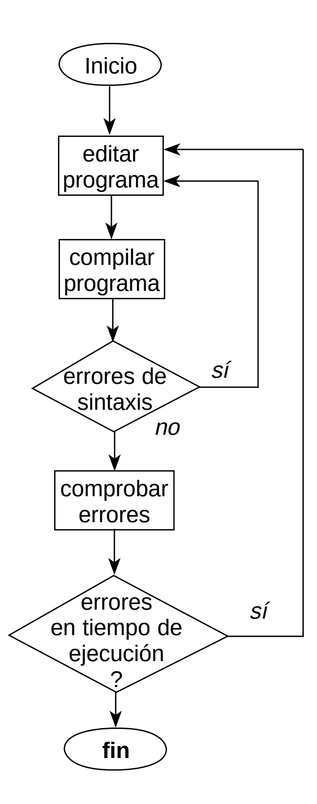


Figura 2.10. Proceso completo de depuración de un programa.

Existen diferentes situaciones en las cuales se suelen introducir errores en un programa. Dos de las más frecuentes son:

1. Violación (no cumplimiento) de las reglas gramaticales del lenguaje de alto nivel en el que se escribe el programa.
2. Los errores en el diseño del algoritmo en el que está basado el programa.

Cuando el compilador detecta un error, visualiza un *mensaje de error* indicando que se ha cometido un error y posible causa del error. Desgraciadamente los mensajes de error son difíciles de interpretar y a veces se llegan a conclusiones erróneas. También varían de un compilador a otro compilador. A medida que se gana en experiencia, el proceso de puesta a punto de un programa se mejora considerablemente. Nuestro objetivo en cada capítulo es describir los errores que ocurren más frecuentemente y sugerir posibles causas de error, junto con reglas de estilo de escritura de programas. Desde el punto de vista conceptual existen tres tipos de errores: *sintaxis, lógicos* y de *regresión*.

2.5.1. Errores de sintaxis

Los **errores de sintaxis** son aquellos que se producen cuando el programa viola la sintaxis, es decir, las reglas de gramática del lenguaje. Errores de sintaxis típicos son: escritura incorrecta de palabras reservadas, omisión de signos de puntuación (comillas, punto y coma...). Los errores de sintaxis son los más fáciles de fijar, ya que ellos son detectados y aislados por el compilador.

Estos errores se suelen detectar por el compilador durante el proceso de compilación. A medida que se produce el proceso de traducción del código fuente (por ejemplo, programa escrito en C++) a lenguaje máquina de la computadora, el compilador verifica si el programa que se está traduciendo cumple las

reglas de sintaxis del lenguaje. Si el programa viola alguna de estas reglas, el compilador genera un *mensaje de error* (o *diagnóstico*) que explica el problema (aparente). Algunos **errores típicos** (ya citados anteriormente):

- Puntos y coma después de la cabecera `main()`.
- Olvido del caracter llave de apertura o cierre (`{`, `}`).
- Omisión de puntos y coma al final de una sentencia.
- Olvido de la doble barra inclinada antes de un comentario.
- Olvido de las dobles comillas al cerrar una cadena.
- ...

Si una sentencia tiene un error de sintaxis no se traducirá completamente y el programa no se ejecutará. Así por ejemplo, si una línea de programa es

```
double radio
```

se producirá un error ya que falta el punto y coma (;) después de la letra última, `"o"`. Posteriormente se explicará el proceso de corrección por parte del programador.

2.5.2. Errores lógicos

Un segundo tipo de error importante es el **error lógico**, ya que tal error representa errores del programador en el diseño del algoritmo y posterior programa. Los errores lógicos son más difíciles de encontrar y aislar, ya que no suelen ser detectados por el compilador.

Suponga, por ejemplo, que una línea de un programa contiene la sentencia

```
double peso = densidad * 5.25 * PI * pow(longitud, 5) / 4.0
```

pero resulta que el tercer asterisco (operador de multiplicación) es en realidad un signo + (operador suma). El compilador no produce ningún mensaje de error de sintaxis, ya que no se ha violado ninguna regla de sintaxis y, por tanto, *el compilador no detecta error* y el programa se compilará y ejecutará bien, aunque producirá resultados de valores incorrectos, ya que la fórmula utilizada para calcular el peso contiene un error lógico.

Una vez que se ha determinado que un programa contiene un error lógico (si es que se encuentra en la primera ejecución y no pasa desapercibida al programador), encontrar el error es una de las tareas más difíciles de la programación. El **depurador** (*debugger*), un programa de software diseñado específicamente para la detección, verificación y corrección de errores, ayudará en las tareas de depuración.

Los errores lógicos ocurren cuando un programa es la implementación de un algoritmo defectuoso. Dado que los errores lógicos normalmente no producen errores en tiempo de ejecución y no visualizan mensajes de error, son más difíciles de detectar porque el programa parece ejecutarse sin contratiempos. El único signo de un error lógico puede ser la salida incorrecta de un programa. La sentencia

```
total_grados_centígrados = fahrenheit_a_centigrados * temperatura_cen;
```

es una sentencia perfectamente legal en C++, pero la ecuación no responde a ningún cálculo válido para obtener el total de grados centígrados en una sala.

Se pueden detectar errores lógicos comprobando el programa en su totalidad, comprobando su salida con los resultados previstos. Se pueden prevenir errores lógicos con un estudio minucioso y detallado del algoritmo antes de que el programa se ejecute, pero resultará fácil cometer errores lógicos y es el conocimiento de C++, de las técnicas algorítmicas y la experiencia lo que permitirá la detección de los errores lógicos.

2.5.3. Errores de regresión

Los **errores de regresión** son aquellos que se crean accidentalmente cuando se intenta corregir un error lógico. Siempre que se corrige un error se debe comprobar totalmente la exactitud (corrección) para asegurarse que se fija el error que se está tratando y no produce otro error. Los errores de regresión son comunes, pero son fáciles de leer y corregir. Una ley no escrita es que: «un error se ha producido, probablemente, por el último código modificado».

2.5.4. Mensajes de error

Los compiladores emiten mensajes de error o de advertencia durante las fases de compilación, de enlace o de ejecución de un programa.

Los mensajes de error producidos durante la compilación se suelen producir, normalmente, por errores de sintaxis y suele variar según los compiladores; pero, en general, se agrupan en tres bloques:

- **Errores fatales.** Son raros. Algunos de ellos indican un error interno del compilador. Cuando ocurre un error fatal, la compilación se detiene inmediatamente, se debe tomar la acción apropiada y, a continuación, se vuelve a iniciar la compilación.
- **Errores de sintaxis.** Son los errores típicos de sintaxis, errores de línea de órdenes y errores de acceso a memoria o disco. El compilador terminará la fase actual de compilación y se detiene.
- **Advertencias** (*warning*). No impiden la compilación. Indican condiciones que son sospechosas, pero son legítimas como parte del lenguaje.

2.5.5. Errores en tiempo de ejecución

Los errores de ejecución se deben, normalmente, a un error en el algoritmo que resuelve el problema. Un ejemplo típico es una división de enteros cuyo código está bien escrito, pero si el divisor es cero se produce un error de ejecución «fatal» que detiene el programa.

Existen dos tipos de *errores en tiempo de ejecución*: aquellos que son detectados por el sistema en tiempo de ejecución del programa C++ —una vez traducido— y aquellos que permiten la terminación del programa pero producen resultados incorrectos.

Un error en tiempo de ejecución puede tener como resultado que el programa obligue a la computadora a realizar una operación ilegal tal como dividir un número por cero o manipular datos no válidos o no definidos. Cuando ocurre este tipo de error, la computadora detendrá la ejecución de su programa y emitirá (visualizará) un mensaje de diagnóstico tal como:

```
Divide error, line number ***
```

Si se intenta manipular datos no válidos o indefinidos, su salida puede contener resultados extraños. Por ejemplo, se puede producir un *desbordamiento aritmético* cuando un programa intenta almacenar un número que es mayor que el tamaño máximo que puede manipular su computadora.

El programa `depurar.cpp` se compila con éxito; pero no contiene ninguna sentencia que asigne un valor a la variable `x` que pueda sumarse a `y` para producir una variable `z`, por tanto, al ejecutarse la sentencia de asignación

```
z = x + y;
```

se produce un error en tiempo de ejecución

```
1:   // archivo depurar
2:   // prueba de errores en tiempo de ejecución
```

```

#include <iostream>
using namespace std;
void main ()
{
   // Datos locales
   float x, y, z;

   y = 10.0
   z = x + y;       // error de ejecución
   cout << "El valor de z es = " << z << endl;
}
```

El programa anterior, sin embargo, podría terminar su ejecución, aunque produciría resultados incorrectos. Dado que no se asigna ningún valor a `x`, contendrá un valor impredecible y el resultado de la suma será también impredecible. Muchos compiladores inicializan las variables automáticamente a cero, haciendo en este caso más difícil de detectar la omisión, sobre todo cuando el programa se transfiere a otro compilador que no asigna ningún valor definido.

Otra fuente de errores en tiempo de ejecución se suele producir por errores en la entrada de datos producidos por la lectura del dato incorrecto en una variable u objeto de entrada.

2.5.6. Pruebas

Los **errores de ejecución** ocurren después de que el programa se ha compilado con éxito y aún se está ejecutando. Existen ciertos errores que la computadora *sólo* puede detectar cuando se ejecuta el programa. La mayoría de los sistemas informáticos detectarán ciertos errores en tiempo de ejecución y presentarán un mensaje de error apropiado. Muchos errores en tiempo de ejecución tienen que ver con los cálculos numéricos. Por ejemplo, si la computadora intenta dividir un número por cero, se produce un error en tiempo de ejecución.

Es preciso tener presente que el compilador puede no emitir ningún mensaje de error durante la ejecución, y eso no garantiza que el programa sea correcto. Recuerde *que el compilador sólo le indica si se escribió bien sintácticamente un programa en C++. No indica si el programa hace lo que realmente desea que haga*. Los errores lógicos pueden aparecer —y de hecho aparecerán— por un mal diseño del algoritmo y posterior programa.

Para determinar si un programa contiene un error lógico, se debe ejecutar utilizando datos de muestra y comprobar la salida verificando su exactitud. Esta **prueba** (*testing*) se debe hacer varias veces utilizando diferentes entradas, preparadas —en el caso ideal— por personas diferentes al programador, que puedan indicar suposiciones no evidentes en la elección de los datos de prueba. Si *cualquier* combinación de entradas produce salida incorrecta, entonces el programa contiene un error lógico.

Una vez que se ha determinado que un programa contiene un error lógico, la localización del error es una de las partes más difíciles de la programación. La ejecución se debe realizar paso a paso (seguir la *traza*) hasta el punto en que se observe que un valor calculado difiere del valor esperado. Para simplificar este *seguimiento* o *traza*, la mayoría de los compiladores de C++ proporcionan un depurador integrado[2] incorporado con el editor, y todos ellos en un mismo paquete de software, que permiten al programador ejecutar realmente un programa, línea a línea, observando los efectos de la ejecución de cada línea en los valores de los objetos del programa. Una vez que se ha localizado el error, se utilizará el editor de texto para corregir dicho error.

Es preciso hacer constar que casi nunca será posible comprobar un programa para todos los posibles conjuntos de datos de prueba. Existen casos en desarrollos profesionales en los que, aparentemente, los

---

[2] Éste es el caso de Borland C++, Builder C++ de Borland/Inprise, Visual C++ de Microsoft o los compiladores bajo UNIX y Linux. Suelen tener un menú `Debug` o bien una opción `Debug` en el menú Run.

programas han estado siendo utilizados sin problemas durante años, hasta que se utilizó una combinación específica de entradas y ésta produjo una salida incorrecta debida a un error lógico. El conjunto de datos específicos que produjo el error nunca se había introducido.

A medida que los programas crecen en tamaño y complejidad, el problema de las pruebas se convierte en un problema de dificultad cada vez más creciente. No importa cuántas pruebas se hagan: «las pruebas nunca se terminan, sólo se detienen, y no existen garantías de que se han encontrado y corregido todos los errores de un programa». Dijkstra ya predijo a principios de los setenta una máxima que siempre se ha de tener presente en la construcción de un programa: «*Las pruebas sólo muestran la presencia de errores, no su ausencia. No se puede probar que un programa es correcto (exacto) sólo se puede mostrar que es incorrecto*».

2.6. LOS ELEMENTOS DE UN PROGRAMA EN C++

Un programa C++ consta de uno o más archivos. Un archivo es traducido en diferentes fases. La primera fase es el preprocesado, que realiza la inclusión de archivos y la sustitución de macros. El preprocesador se controla por directivas introducidas por líneas que contienen # como primer carácter. El resultado del preprocesado es una secuencia de *tokens.*

2.6.1. Tokens (elementos léxicos de los programas)

Existen cinco clases de *tokens*: identificadores, palabras reservadas, literales, operadores y otros separadores.

2.6.2. Identificadores

Un *identificador* es una secuencia de caracteres, letras, dígitos y subrayados (_). El primer carácter debe ser una letra (puede ser un subrayado) en ANSI/ISO (++). Las letras mayúsculas y minúsculas son diferentes a efectos del identificador.

```
nombre_clase     Indice           Día_Mes_Año
elemento_mayor   Cantidad_Total   Fecha_Compra_Casa
a                Habitacion120    i
```

En C++ el identificador puede ser de cualquier longitud; sin embargo la mayoría de los compiladores pueden imponerle alguna restricción; en cualquier caso le recomendamos no sea muy largo pensando en la escritura y posibles modificaciones.

C++ es *sensible a las mayúsculas*, por consiguiente, C++ reconoce como distintos los identificadores `ALFA` y `alfa` (le recomendamos que utilice siempre el mismo estilo al escribir sus identificadores). Un consejo que puede servir de posible regla puede ser:

1. Escribir identificadores de variables en letras minúsculas.
2. Constantes en mayúsculas.
3. Funciones con tipo de letra mixto: mayúscula/minúscula.

Reglas básicas de formación de identificadores

1. Secuencia de letras o dígitos; el primer carácter puede ser una letra o un subrayado (compiladores de Borland, entre otros).
2. Los identificadores son sensibles a las mayúsculas y minúsculas:

   `minum` es distinto de `MiNum`

3. Los identificadores pueden tener cualquier longitud, pero sólo son significativos los 32 primeros (ése es el caso de Borland y Microsoft).
4. Los identificadores no pueden ser palabras reservadas, tales como `if`, `switch` o `else`.

2.6.3. Palabras reservadas

Una palabra reservada (*keyword* o *reserved word*), tal como **void** es una característica del lenguaje C++ asociada con algún significado especial. Una palabra reservada no se puede utilizar como nombre de identificador, objeto o función.

```
// ...
void void()      // error
{
  // ...
  int char;      // error
  // ...
}
```

Los siguientes identificadores están reservados para utilizarlos como *palabras reservadas*, y no se deben emplear para otros propósitos.

```
asm         double      mutable     struct
auto        else        namespace   switch
bool        enum        new         template
break       explicit    operator    this
case        extern      private     throw
catch       float       protected   try
char        for         public      typedef
class       friend      register    union
const       goto        return      unsigned
continue    if          short       virtual
default     inline      signed      void
delete      int         sizeof      volatile
do          long        static      wchar_t
                                    while
```

2.6.4. Signos de puntuación y separadores

Todas las sentencias deben terminar con un punto y coma. Otros signos de puntuación son:

```
! % ^ & * ( ) - + = { } ~
[ ] \ ; ´ : < > ? , . / "
```

Los separadores son espacios en blanco, tabulaciones, retornos de carro y avances de línea.

2.6.5. Archivos de cabecera

Un archivo de cabecera es un archivo especial que contiene las declaraciones de objetos y funciones de la biblioteca. Para utilizar objetos y funciones almacenadas en una biblioteca, un programa debe utilizar la directiva `#include` para insertar el archivo de cabecera correspondiente. Por ejemplo, si un

programa utiliza la función pow que se almacena en la biblioteca matemática *math.h*, debe contener la directiva

```
#include <math.h>
```

para hacer que el contenido de la biblioteca matemática esté disponible a un programa.

La mayoría de los programas contienen líneas como ésta al principio, que se incluyen en el momento de compilación.

```
#include <iostream.h>     #include "iostream.h"
```

2.6.6. El estándar C++ (ANSI C++)

El estándar ANSI C++ ha cambiado el convenio (notación) de los archivos de cabecera. Es posible utilizar sólo los nombres de las bibliotecas sin el sufijo .h; es decir, se puede usar iostream, cmath, cassert y cstdlib en lugar de iostream.h, math.h, assert.h y stdlib.h, respectivamente. En ANSI C++ se puede utilizar indistintamente.

```
#include <iostream.h>     o bien     #include <iostream>
```

Sin embargo, no se puede utilizar

```
#include <string>
```

ya que no es lo mismo que

```
#include <string.h>
```

La razón es que la cabecera <string> define el estilo C++ para cadenas y la cabecera <string.h> define cadenas estilo C que son totalmente diferentes.

2.7. TIPOS DE DATOS EN C++

C++ no soporta un gran número de tipos de datos predefinidos, pero tiene la capacidad para crear sus propios tipos de datos. Todos los tipos de datos simples o básicos de C++ son, esencialmente, números. Los tres tipos de datos básicos son:

- enteros;
- números de coma flotante (*reales*);
- caracteres.

La Tabla 2.3 recoge los principales tipos de datos básicos, sus tamaños en bytes y el rango de valores que puede almacenar.

Los tipos de datos fundamentales en C++ son:

- **enteros** (números completos y sus negativos): de tipo int.
- **variantes de enteros:** tipos short, long y unsigned.
- **reales:** números decimales: tipos float, double o long double.
- **caracteres:** letras, dígitos, símbolos y signos de puntuación.

Tabla 2.3. Tipos de datos simples de C++.

Tipo	Ejemplo	Tamaño en bytes	Rango Mínimo..Máximo
`char`	`'C'`	1	`0..255`
`short`	`-15`	2	`-128..127`
`int`	`1024`	2	`-32768..32767`
`unsigned int`	`42325`	2	`0..65535`
`long`	`262144`	4	`-2147483648..2147483637`
`float`	`10.5`	4	$3.4*(10^{-38})..3.4*(10^{38})$
`double`	`0.00045`	8	$1.7*(10^{-308})..1.7*(10^{308})$
`long double`	`1e-8`	8	*igual que* `double`

`char`, `int`, `float` y `double` son palabras reservadas, o más específicamente, *especificadores de tipos*. Cada tipo de dato tiene su propia *lista de atributos* que definen las características del tipo y pueden variar de una máquina a otra. Los tipos `char`, `int` y `double` tienen variaciones o *modificadores de tipos de datos*, tales como `short`, `long`, `signed` y `unsigned`, para permitir un uso más eficiente de los tipos de datos.

Un tipo carácter adicional añadido en ANSI C++ es `wchar_t`:

`wchar_t` constante carácter ancha *(wide)*

Existen dos tipos adicionales en C++ que se tratarán más adelante:

`enum` constante de enumeración
`bool` constante *falso-verdadero*
`void` tipo especial de dato (ausencia de tipo)

2.7.1. Enteros (`int`)

Probablemente el tipo de dato más familiar es el entero, o tipo `int`. Los enteros son adecuados para aplicaciones que trabajen con datos numéricos. Los tipos enteros se almacenan internamente en 2 bytes (o 16 bits) de memoria. La Tabla 2.4 resume los tres tipos enteros básicos, junto con el rango de valores y el tamaño en bytes usual, dependiendo de cada máquina.

Tabla 2.4. Tipos de datos enteros.

Tipo C++	Rango de valores	Uso recomendado
`int`	`-32.768 .. +32.767`	Aritmética de enteros, bucles `for`, conteo.
`unsigned int`	`0 .. 65.535`	Conteo, bucles `for`, índices.
`short int`	`-32.768 .. +32.767`	Aritmética de enteros, bucles `for`, conteo.

Declaración de variables

La forma más simple de una declaración de variable en C++ es poner primero el tipo de dato y, a continuación, el nombre de la variable. Si se desea dar un valor inicial a la variable, éste se pone a continuación. El formato de la declaración es:

```
<tipo de dato> <nombre de variable> = <valor inicial>
```

Se pueden también declarar múltiples variables en la misma línea:

```
<tipo_de_dato> <nom_var1>, <nom_var2> ... <nom-varn>
```

Así, por ejemplo,

```
int valor;           int valor = 99;
int valor1, valor2;  int num_parte = 1141, num_items = 45;
```

Los tres modificadores (`unsigned`, `short`, `int`) que funcionan con `int` (Tabla 2.4) varían el rango de los enteros.

En aplicaciones generales, las constantes enteras se pueden escribir en *decimal* o *base 10*; por ejemplo, `100`, `200` o `450`. Para escribir una constante sin signo, se añade la letra `U` (o bien `u`). Por ejemplo, para escribir `40.000`, escriba `40000U`.

Si se utiliza C++ para desarrollar software para sistemas operativos o para hardware de computadora, C++ permite escribir constantes enteros en *octal* (base 8) o *hexadecimal* (base 16). Una constante octal es cualquier número que comienza con un 0 y contiene dígitos en el rango de 1 a 7. Por ejemplo, 0377 es un número octal. Una constante hexadecimal comienza con 0x y va seguida de los dígitos 0 a 9 o las letras `A` a `F` (o bien `a` a `f`). Por ejemplo, `0xFF16` es una constante hexadecimal.

La Tabla 2.5 muestra ejemplos de constantes enteras representadas en sus notaciones (bases) decimal, hexadecimal y octal.

Tabla 2.5. Constantes enteras en tres bases diferentes.

Base 10 **Decimal**	**Base 16** **Hexadecimal (Hex)**	**Base 8** **Octal**
8	0x08	010
10	0x0A	012
16	0x10	020
65536	0x10000	0200000
24	0x18	030
17	0x11	021

Cuando el rango de los tipos enteros básicos no es suficientemente grande para sus necesidades, se consideran tipos enteros largos. La Tabla 2.6 muestra los dos tipos de datos enteros largos. Ambos tipos requieren 4 bytes de memoria (32 bits) de almacenamiento.

Tabla 2.6. Tipos de datos enteros largos.

Tipo C++	**Rango de valores**
long	-2.147.483.648 .. 2.147.483.647
unsigned long	0 .. 4.294.967.295

Un ejemplo de uso de enteros largos es:

```
long medida_milimetros;

unsigned long distancia_media;
```

Si se desea forzar al compilador para tratar sus constantes como `long`, añada la letra `L` (o bien `l`) a su constante. Por ejemplo,

```
long numeros_grandes = 40000L;
```

2.7.2. Tipos de coma flotante (`float/double`)

Los tipos de datos de coma (*punto*) flotante representan números reales que contienen una coma (un punto) decimal, tal como `3.14159`, o números muy grandes, tales como `1.85*`10^{15}.

La declaración de las variables de coma flotante es igual que la de variables enteras. Así, un ejemplo es

```
float valor;            //declara una variable real
float valor1, valor2;   //declara varios valores de coma
                        //flotante
float valor = 99.99;    //asigna el valor 99.99 a la
                        //variable valor
```

C++ soporta tres formatos de coma flotante (Tabla 2.7). El tipo `float` requiere 4 bytes de memoria, `double` requiere 8 bytes y `long double` requiere 10 bytes.

Tabla 2.7. Tipos de datos en coma flotante (Borland C++).

Tipo C++	Rango de valores	Precisión	
float	3.4 x 10^{-38} ...	3.4 x 10^{38}	7 dígitos
double	1.7 x 10^{-308} ...	1.7 x 10^{308}	15 dígitos
long double	3.4 x 10^{-4932} ...	1.1 x 10^{4932}	19 dígitos

Ejemplos

```
float f;                        // definición de f
f = 5.65                        // asignación
cout << "f:" << f << endl;      // visualización de f:5.65
double h;                       // definición de h
h = 0.0                         // asignación
cout << "h:" << h << endl;      // visualiza h: 0.0
```

2.7.3. Caracteres (`char`)

Un *carácter* es cualquier elemento de un conjunto de caracteres predefinidos o alfabeto. La mayoría de las computadoras utilizan el conjunto de caracteres ASCII (véase Apéndice D, en página web `www.mh.es/joyanes`).

C++ procesa datos carácter (tales como texto) utilizando el tipo de dato `char`. En unión con la estructura *array*, que se verá posteriormente, se puede utilizar para almacenar *cadenas de caracteres* (grupos de caracteres). Se puede definir una variable carácter escribiendo

```
char dato_car;
char letra = 'A';
char respuesta = 'S';
```

Internamente, los caracteres se almacenan como números. La letra `A`, por ejemplo, se almacena internamente como el número `65`, la letra `B` es `66`, la letra `C` es `67`, etc. El tipo char representa valores en el rango `–128` a `+127` y se asocian con el código ASCII.

Dado que el tipo `char` almacena valores en el rango de –128 a +127, C++ proporciona el tipo `unsignedchar` para representar valores de 0 a 255.

Puesto que los caracteres se almacenan internamente como números, se pueden realizar operaciones aritméticas con datos tipo `char`. Por ejemplo, se puede convertir una letra minúscula *a* en una letra mayúscula *A*, restando 32 del código ASCII (*véase* Apéndice D: **código ASCII** en página web del libro). Así, para realizar la conversión, restar 32 del tipo de dato `char`, como sigue:

```
char car_uno = 'a';
...
car_uno = car_uno - 32;
```

Esto convierte a (código ASCII 97) a A (código ASCII 65). De modo similar, añadiendo 32 convierte el carácter de letra mayúscula a minúscula:

```
car_uno = car_uno + 32;
```

Como los tipos `char` son subconjuntos de los tipos enteros, se puede asignar un tipo char a un entero. Por ejemplo,

```
int suma = 0;
char valor;
...
cin >> valor;           // operador de entrada; apartado
                        // 2.8.2
suma = suma + valor;    // operador...
cout << suma;
```

Existen caracteres que tienen un propósito especial y no se pueden describir utilizando el método normal. C++ proporciona **secuencias de escape**. Por ejemplo, el literal carácter de un apóstrofe se puede escribir como

```
'\"
```

y el carácter nueva línea

```
'\n'
```

La Tabla 2.8 (página 85) enumera las diferentes secuencias de escape de C++.

2.7.5. Tipo `void`

El tipo `void` indica que no toma ningún valor (nada). Este tipo que aparentemente no hace nada, se utiliza —como se verá más adelante— en C++ para algunas aplicaciones, tales como:

1. Declarar una función que no va a tener parámetros.
2. Declarar punteros genéricos `void`.

2.8. EL TIPO DE DATO BOOL

El tipo `bool`[3] se suele utilizar para indicar si ha ocurrido o no un suceso. También para efectuar comparaciones. Así, por ejemplo, si una variable `puerta` representa el hecho de que una puerta esté abierta o

---

[3] El tipo `bool` proviene del matemático inglés, del siglo XIX, George Boole, que inventó el concepto de utilizar operadores lógicos con valores verdadero-o-falso. Tales valores se les suele conocer en su honor como valores *booleans* o *booleanos*.

cerrada, puede tomar el valor *falso* cuando está cerrada y *verdadero* cuando está abierta. Otros casos pueden ser: si una bombilla está encendida o apagada, si hay clase o no hay clase a una determinada hora, etc.

Los compiladores de C++ que siguen la norma ANSI incorporan un nuevo tipo de dato ***`bool`*** cuyos valores son: *verdadero* (*true*) y *falso* (*false*). Las expresiones lógicas devuelven en estos compiladores valores de este tipo, en lugar de valores tradicionales de tipo `int` que es el sistema estilo C que siguen los compiladores antiguos de C++. El tipo **`bool`** proporciona la capacidad de declarar variables lógicas, que pueden almacenar los valores verdadero y falso.

Ejemplo

```
bool bisiesto;
bisiesto = true;
bool encontrado, bandera;
```

Dadas estas declaraciones, las siguientes asignaciones son todas válidas:

```
encontrado = true;        // encontrado toma el valor
                          // verdadero
indicador = false;        // indicador toma el valor falso
encontrado = indicador;   // encontrado toma el valor de
                          // indicador
```

Ejemplo

```
// bool.cpp
// muestra uso de bool

#include <iostream>
using namespace std;

void main()
{
  bool b1;
  bool b2 = false;
  bool b3 = true;
  int i1 = true;
  int i2 = false;
  int i3 = b3;
  int i4 = 10;
  bool b4 = i4;
  bool b5 = -i4;
  // error, no se puede asignar a literales true-false
  true = 2;
  false = -1;
  cout << "b1:" << b1 << endl;
  ...
}
```

2.8.1. Simulación del tipo `bool`

Si su compilador C++ no incluye el tipo `bool`, deberá utilizar el tipo de dato `int` para representar el tipo de dato `bool`. C++ utiliza el valor entero 0 para representar falso y cualquier valor entero distinto de cero (normalmente 1) para representar verdadero. De esta forma, se pueden utilizar enteros para escribir expresiones lógicas de igual forma que se utiliza el tipo `bool`. Una expresión lógica que se evalúa a «0» se considera falsa; una expresión lógica que se evalúa a «distinto de cero» se considera verdadera.

`Valor distinto de cero`	representa	*true* (verdadero)
`0`	representa	*false* (falso)

Antes de que se introdujera el tipo `bool` en C++, el sistema usual de declarar datos lógicos era definir un tipo enumerado `Boolean` con dos valores *false* y *true* de la forma siguiente:

```
enum Boolean  { FALSE, TRUE };
```

Esta declaración hace a `Boolean` un tipo definido por el usuario con literales (valores constantes) `TRUE` y `FALSE`.

Ejercicio 2.1

Si desea simular el tipo `bool` pero al estilo de tipo incorporado propio, se podría conseguir construyendo un archivo `.h (boolean)` con constantes con nombre TRUE y FALSE, tal como

```
// archivo: boolean.h
#ifndef BOOLEAN_H
#define BOOLEAN_H
typedef int Boolean;
const int TRUE = 1;
const int FALSE = 0;
#endif   // BOOLEAN_H
```

Entonces, basta con incluir el archivo `"boolean.h"` y utilizar `Boolean` como si fuera un tipo de dato incorporado con los literales `TRUE` y `FALSE` como literales lógicos o booleanos.

Si desea utilizar las letras minúsculas para definir `bool`, `true` y `false` con compiladores antiguos, se puede utilizar esta versión del archivo de cabecera `boolean.h`.

```
// archivo: boolean.h
#ifndef BOOLEAN_H
#define BOOLEAN_H
typedef int bool;
const int true = 1;
const int false = 0;
#endif  // BOLEAN_H
```

2.8.2. Escritura de valores `bool`

La mayoría de las expresiones lógicas aparecen en estructuras de control que sirven para determinar la secuencia en que se ejecutan las sentencias C++. Raramente se tiene la necesidad de leer valores `bool` como dato de entrada o de visualizar valores `bool` como resultados de programa. Si es necesario, se puede visualizar el valor de la variable `bool` utilizando el operador de salida <<. Así, si `bandera` es `false`, la sentencia

```
cout << "El valor de bandera es " << Bandera;
```

visualizará

```
El valor de bandera es 0
```

Si se necesita leer un dato de una variable tipo `bool`, se puede representar el dato como un entero: usar 0 para falso y 1 para verdadero.

2.9. CONSTANTES

Las constantes se pueden declarar con la palabra reservada `const` y se les asigna un valor en el momento de la declaración; este valor no se puede modificar durante el programa y cualquier intento de alterar el valor de un identificador definido con el calificador `const` producirá un mensaje de error del calificador.

Las constantes no cambian durante la ejecución del programa. En C++ existen cuatro tipos de constantes:

- *constantes literales*,
- *constantes definidas*,
- *constantes enumeradas*,
- *constantes declaradas.*

Las constantes literales son las más usuales; toman valores tales como `45.32564,222` o bien `"Introduzca sus datos"` que se escriben directamente en el texto del programa. Las constantes definidas son identificadores que se asocian con valores literales constantes y que toman determinados nombres. Las constantes declaradas son como variables: sus valores se almacenan en memoria, pero no se pueden modificar. Las constantes enumeradas permiten asociar un identificador, tal como `Color`, con una secuencia de otros nombres, tales como `Azul`, `Verde`, `Rojo` y `Amarillo`.

2.9.1. Constantes literales

Las constantes literales o *constantes*, en general, se clasifican también en cuatro grupos, cada uno de los cuales puede ser de cualquiera de los tipos:

- constantes enteras,
- constantes caracteres,
- constantes de coma flotante,
- constantes de cadena.

Constantes enteras

La escritura de constantes enteras requiere seguir unas determinadas reglas. *Recuerde*:

- No utilizar nunca comas ni otros signos de puntuación en números enteros o completos.

  `123456` *en lugar de* `123.456`

- Para forzar un valor al tipo `long`, terminar con una letra `L` mayúscula. Por ejemplo,

  `1024` *es un tipo entero* `1024L` *es un tipo largo* `(long)`

- Para forzar un valor al tipo `unsigned`, terminarlo con una letra mayúscula `U`. Por ejemplo, `4352U`.

Formato decimal	`123`
Formato octal	`0777` (están precedidas de la cifra `0`)
Formato hexadecimal	`0XFF3A` (están precedidas de `"Ox"` o bien, `"OX"`)

Se pueden combinar sufijos `L(l)`, que significa *long* (largo), o bien, `U(u)`, que significa *unsigned* (sin signo).

```
3456UL
```

Constantes reales

Una constante flotante representa un número real; siempre tienen signo y representan aproximaciones en lugar de valores exactos.

```
82.  347  .63  83.  47e-4  1.25E7  61.e+4
```

Se pueden escribir constantes de coma flotante de diversas formas. Si un número es entero se puede escribir con o sin punto decimal:

```
15
```

o bien

```
15.0
```

Si el valor de coma flotante tiene una parte decimal, se puede escribir ésta después del punto decimal, como

```
3.141519       –3.151519
```

Para escribir números en notación exponencial, se debe seguir la parte decimal del número con la letra `E` (o bien `e`) y, a continuación, el exponente. Por ejemplo,

```
4.5E+5    -3.2E-5    7.12E6
```

La notación científica se representa con un exponente positivo o negativo.

2.5E4	*equivale a*	25000
5.435E-3	*equivale a*	0.005435

Existen tres tipos de constantes:

`float`	*4 bytes*
`double`	*8 bytes*
`long double`	*10 bytes*

Constantes carácter

Una constante carácter (`char`) es un carácter del código ASCII encerrado entre comillas simples.

```
'A'    'b'    'c'
```

Además de los caracteres ASCII estándar, una constante carácter soporta caracteres especiales que no se pueden representar utilizando su teclado, como por ejemplo, los códigos ASCII altos y las secuencias de escape. (El Apéndice D, de la página web del libro, recoge un listado de todos los caracteres ASCII.)

Así, por ejemplo, el carácter sigma (M) —código ASCII 228, Hex (Hexadecimal `E4`— se representa mediante el prefijo `\x` y el número hexadecimal del código ASCII. Por ejemplo,

```
char sigma = '\xE4';
```

Este método se utiliza para almacenar o imprimir cualquier carácter de la tabla ASCII por su número hexadecimal. En el ejemplo anterior, la variable sigma no contiene cuatro caracteres sino únicamente el símbolo sigma.

Un carácter que se lee utilizando una barra oblicua (\) se llama *secuencia* o *código de escape*. La Tabla 2.8 muestra diferentes secuencias de escape y su significado.

```
// Programa: Pruebas códigos de escape
#include <iostream>
using namespace std;

main()
{
  char alarma = '\a';       //alarma
  char bs = '\b';           //retroceso de espacio
  cout << alarma;
  cout << bs;
  return 0;
}
```

Tabla 2.8. Caracteres secuenciales (códigos) de escape.

Código de Escape	**Significado**	**Códigos ASCII**	
		Dec	**Hex**
'\n'	nueva línea	13 10	0D 0A
'\r'	retorno de carro	13	0D
'\t'	Tabulación	9	09
'\v'	tabulación vertical	11	0B
'\a'	alerta (pitido sonoro)	7	07
'\b'	retroceso de espacio	8	08
'\f'	avance de página	12	0C
'\\'	barra inclinada inversa	92	5C
'\''	comilla simple	39	27
'\"'	doble comilla	34	22
'\?'	signo de interrogación	34	22
'\000'	número octal	*Todos*	*Todos*
'\xhh'	número hexadecimal	*Todos*	*Todos*

Declaración de constantes: `#define`

La creación de constantes se puede realizar con la palabra reservada `const` y la directiva `#define`.

Estilo de escritura

Los identificadores de constantes se escriben con mayúsculas y los identificadores de variables con minúsculas.

Las constantes declaradas mediante `#define` se suelen escribir antes de la función `main` y después de la directiva `#include`.

Directiva `#define`

Se pueden asignar cadenas a las constantes creadas con `#define`

```
#define   SALUDO   "Buenos dias"
#define   GRUPO    "FM-11"
```

Aritmética con caracteres C++

Dada la correspondencia entre un carácter y su código ASCII, es posible realizar operaciones aritméticas sobre datos de caracteres. Observe el siguiente segmento de código:

```
char c;
c = 'T' + 5;       // suma 5 al carácter ASCII
```

Realmente lo que sucede es almacenar `Y` en `c`. El valor ASCII de la letra `T` es 84, y al sumarle 5 produce 89, que es el código de la letra `Y`. A la inversa, se pueden almacenar constantes de carácter en variables enteras. Así,

```
int j = 'p'
```

No pone una letra `p` en `j`, sino que asigna el valor 80 (código ASCII de `p`) a la variable `j`.

Constantes cadena

Una *constante cadena* (también llamada *literal cadena* o simplemente *cadena*) es una secuencia de caracteres encerrados entre dobles comillas. Algunos ejemplos de constantes de cadena son:

```
"123"
"12 de octubre 1492"
"esto es una cadena"
```

Se puede escribir una cadena en varias líneas, terminando cada línea con `"\"`

```
"esto es una cadena\
 que tiene dos lineas"
```

Se puede concatenar cadenas, escribiendo

```
"ABC"  "DEF"  "GHI"
"JKL"
```

que equivale a

```
"ABCDEFGHIJKL"
```

En memoria, las cadenas se representan por una serie de caracteres ASCII más un 0 o nulo. El carácter nulo marca el final de la cadena y se inserta automáticamente por el compilador C++ al final de las constantes de cadenas. Para representar valores nulos, C++ define el símbolo `NULL` como una constante en diversos archivos de cabecera (normalmente `stdef.h`, `stdio.h`, `stdlib.h` y `string.h`). Para utilizar `NULL` en un programa, incluya uno o más de estos archivos en lugar de definir `NULL` con una línea tal como

```
#define NULL 0
```

Recuerde que una constante de caracteres se encierra entre comillas simples, y las constantes de cadena encierran caracteres entre dobles comillas. Por ejemplo,

```
'Z'    "Z"
```

El primer `'Z'` es una constante carácter simple con una longitud de 1, y el segundo `"Z"` es una constante de cadena de caracteres también con la longitud 1. La diferencia es que la constante de cadena incluye un cero nulo al final de la cadena, ya que C++ necesita conocer dónde termina la cadena. Por consiguiente, *no puede mezclar constantes caracteres y cadenas de caracteres en su programa*.

2.9.2. Constantes definidas (simbólicas): `#define`

Las constantes pueden recibir nombres simbólicos mediante la directiva `#define`.

```
#define NUEVALINEA '\n'
#define PI 3.141592   //valor de Pi
#define VALOR 54
```

C++ sustituye los valores `\n`, `3.141592` y `54` cuando se encuentra las constantes simbólicas `NUEVALINEA`, `PI` y `VALOR`. Las líneas anteriores no son sentencias y, por ello, no terminan en punto y coma.

```
cout << "El valor es " << VALOR << NUEVALINEA;
```

2.9.3. Constantes enumeradas

Las constantes enumeradas permiten crear listas de elementos afines. Un ejemplo típico es una constante enumerada de lista de colores, que se puede declarar como:

```
enum Colores {Rojo, Naranja, Amarillo, Verde, Azul, Violeta};
```

Cuando se procesa esta sentencia, el compilador asigna un valor que comienza en 0 a cada elemento enumerado; así, `ROJO` equivale a 0, `NARANJA` es 1, etc. El compilador *enumera* los identificadores por usted. Después de declarar un tipo de dato enumerado, se pueden crear variables de ese tipo, como con cualquier otro tipo de datos. Así, por ejemplo, se puede definir una variable de tipo `Colores`.

```
Colores Colorfavorito = Verde;
```

Otro ejemplo puede ser:

```
enum Boolean { False, True };
```

que asignará al elemento `False` el valor 0 y a `True` el valor 1.

Para crear una variable de tipo lógico declarar:

```
Boolean Interruptor = True;
```

Es posible asignar valores distintos de los que les corresponde en su secuencia natural

```
enum LucesTrafico {Verde, Amarillo = 10, Rojo};
```

2.9.4. Constantes declaradas `const` y `volatile`

El cualificador `const` permite dar nombres simbólicos a constantes a modo de otros lenguajes, como Pascal. El formato general para crear una constante es:

```
const tipo nombre = valor;
```

Si se omite *tipo*, C++ utiliza `int` (entero por defecto)

```
const int Meses=12; // Meses es constante simbólica valor 12

const float Pi = 3.141592 //numero Pi
const char CARACTER='@';
const int OCTAL=0233;
const char CADENA []="Curso de C++";
```

C++ soporta el calificador de tipo variable **`const`**. Especifica que el valor de una variable no se puede modificar durante el programa. Cualquier intento de modificar el valor de la variable definida con `const` producirá un mensaje de error.

```
const int semana = 7;
const char CADENA []= "Programación en C++";
```

La palabra reservada **`volatile`** actúa como **`const`**, pero su valor puede ser modificado no sólo por el propio programa, sino también por el *hardware* o por el *software* del sistema. Las variables volátiles, sin embargo, no se pueden guardar en registros, como es el caso de las variables normales.

Diferencias entre `const` y `#define`

Las definiciones `const` especifican tipos de datos, terminan con puntos y coma y se inicializan como las variables. La directiva `#define` no especifica tipos de datos, no utilizan el operador de asignación (=) y no terminan con punto y coma.

Ventajas de `const` sobre `#define`

En C++ casi siempre es recomendable el uso de `const` en lugar de `#define`. Además de las ventajas ya enunciadas se pueden considerar otras:

- El compilador, normalmente, genera código más eficiente con constantes `const`.
- Como las definiciones especifican tipos de datos, el compilador puede comprobar inmediatamente si las constantes literales en las definiciones de `const` están en forma correcta. Con `#define` el compilador no puede realizar pruebas similares hasta que una sentencia utiliza el identificador constante, por lo que se hace más difícil la detección de errores.

Desventaja de `const` sobre `#define`

Los valores de los símbolos de `const` ocupan espacio de datos en tiempo de ejecución, mientras que `#define` sólo existe en el texto del programa y su valor se inserta directamente en el código compilado. Por esta razón, algunos programadores de C siguen utilizando `#define` en lugar de `const`.

Sintaxis de `const`

```
const  tipoDato  nombreConstante = valorConstante;

const unsigned DiasDeSemana = 7;
const HorasDelDia = 24;
```

2.10. VARIABLES

En C++ una *variable* es una posición con nombre en memoria donde se almacena un valor de un cierto tipo de dato y puede ser modificado. Las variables pueden almacenar todo tipo de datos: cadenas, números y estructuras. Una *constante*, por el contrario, es una variable cuyo valor no puede ser modificado.

Una variable típicamente tiene un nombre (un identificador) que describe su propósito. Toda variable utilizada en un programa debe ser declarada previamente. La definición en C++ puede situarse en cualquier parte del programa (en C ANSI debe declararse, sin embargo, al principio del bloque, antes de toda sentencia ejecutable). Una definición reserva un espacio de almacenamiento en memoria. El procedimiento para definir (*crear*) una variable es escribir el tipo de dato, el identificador o nombre de la variable y, en ocasiones, el valor inicial que tomará. Por ejemplo,

```
char Respuesta;
```

significa que se reserva espacio en memoria para `Respuesta`, en este caso, un carácter ocupa un byte.

El nombre de una variable ha de ser un identificador válido. Es frecuente, en la actualidad, utilizar subrayados en los nombres, bien al principio o en su interior, con objeto de obtener mayor legibilidad y una correspondencia mayor con el elemento del mundo real que representa.

```
salario   dias_de_semana   edad_alumno   _fax
```

2.10.1. Declaración

Una *declaración* de una variable es una sentencia que proporciona información de la variable al compilador C++. Su sintaxis es:

```
tipo variable
```

`tipo` *es el nombre de un tipo de dato conocido por C++*
`variable` *es un identificador (nombre) válido en C++.*

Ejemplos

```
long   dNumero;
double HorasAcumuladas;
float  HorasPorSemana;
float  NotaMedia;
short  DiaSemana;
```

Es preciso *declarar* las variables antes de utilizarlas. Se puede declarar una variable en dos lugares dentro de un programa:

- al principio de un archivo o bloque de código;
- en el punto de utilización.

Al principio de un archivo o bloque de código

Éste es el medio tradicional de C para declarar una variable. La variable se declara al principio del archivo en código fuente o bien al principio de una función.

```
#include <iostream>  //variable al principio del archivo
using namespace std;

int MiNumero;

main()
{
  cout << "¿Cuál es su número favorito?";
  cin >> MiNumero;
  return 0;
}
```

```
#include <iostream>  //variable al principio de función
...

main()
{
  int i;
  int j;
  ...
}
```

En el punto de utilización

C++ proporciona una mayor flexibilidad que C en la declaración de variables, ya que es posible declarar una variable en el punto donde se vaya a utilizar. Esta propiedad se utiliza mucho en el diseño de bucles (Capítulo 6). En C el sistema de declarar una variable para controlar el bucle `for` es:

```
int j;
for(j = 0; j < 10; j++)
{
  // ...
}
```

Sin embargo, en C++ es posible declarar la variable en el momento de su utilización.

```
for(int j = 0; j < 10; j++)
{
  // ...
}
```

En C las declaraciones se han de situar siempre al principio del programa, mientras que en C++ las declaraciones se pueden mezclar con sentencias ejecutables. Su ámbito es el bloque en el que están declaradas.

Ejemplo

```
// Distancia a la luna en kilometros
#include <iostream>
using namespace std;

void main()
{
  const int Luna = 238857;//distancia en millas
  cout << "Distancia a la Luna " << luna;
  cout << " millas\n";
  int luna_kilo;
  luna_kilo = luna*1.609//una milla = 1.609 kilómetros
  cout << "En kilómetros es " << luna_kilo;
  cout << " km.\n";
}
```

Ejemplo 2.5

El siguiente programa muestra cómo una variable puede ser declarada en cualquier parte de un programa C++.

```
#include <iostream.h>
using namespace std;

// Diferentes declaraciones
main()
{
```

```
  int x, y1;// declarar a las variables x e y1
  x = 75;
  y1 = 89;
  int y2 = 50;// declarar a variable y2 inicializándola a 50
  cout << x << "," << y1 << "," << y2 << endl;
  return 0;
}
```

2.10.2. Inicialización de variables

Cuando se declara una variable, ésta no tiene ningún valor asociado; eso significa que si se utiliza esa variable sin ningún valor inicial, el programa podría funcionar incorrectamente. Aunque no es obligatoria la inicialización de variables, sí es recomendable.

En algunos programas anteriores, se ha proporcionado un valor denominado **valor inicial**, a una variable cuando se declara. El formato general de una declaración de inicialización es:

> *tipo nombre_variable = expresión*
>
> *expresión* es cualquier expresión válida cuyo valor es del mismo tipo que *tipo*.
>
> *Nota:* esta sentencia declara y proporciona un valor inicial a una variable.

Las variables se pueden inicializar a la vez que se declaran, o bien, inicializarse después de la declaración. El primer método es probablemente el mejor en la mayoría de los casos, ya que se combina la definición de la variable con la asignación de su valor inicial.

```
char respuesta = 'S';   char var1 = 'A';
int contador = 1;       char var2 = '\t';
float peso = 156.45;    char var3 = 'B';
int anio = 1992;        char var4 = 'C';
```

Estas acciones crean variables `respuesta`, `contador`, `peso`, `anio`, `var1`, `var2`, `var3` y `var4`, que almacenan en memoria los valores respectivos situados a su derecha.

El segundo método consiste en utilizar sentencias de asignación diferentes después de definir la variable, como en el siguiente caso:

```
char barra;
barra = '/';
```

2.10.3. Declaración o definición

La diferencia entre *declaración* y *definición* es sutil. Una declaración introduce un nombre de un objeto o de una variable (tal como `c_var`) y asocia un tipo con la variable/objeto tal como `int`. Una definición es una declaración que asigna simultáneamente memoria al objeto/variable.

Una declaración de una variable sirve para tres propósitos:

- Define el nombre de la variable.
- Define el tipo de la variable (entero, real, carácter, etc.).
- Proporciona al programador una descripción de la variable.

El formato general de una declaración de variables es:

```
tipo   nombre;      //comentario, opcional
 |       |
 |     identificador válido
int, float, ...

double x;    // declara el nombre de la variable x de tipo
             // double
char c_var;  // declara c_var de tipo char
int i;       // definido pero no inicializado
int i = 0;   // definido e inicializado a cero.
```

Normalmente, las declaraciones suelen ser también definiciones, pero existen diferentes tipos de declaraciones que no son definiciones.

Declaraciones *versus* Definiciones

C++ hace una distinción entre *declaración* y *definición* de una variable (o identificador). Cuando *se declara* una variable (identificador) se introduce el nombre. Cuando *se define* se asigna a almacenamiento al elemento nombrado. En el caso de los tipos de variables que se examinan en este capítulo y siguientes, el concepto *declaración de una variable* representa la declaración y la definición; es decir, se asigna almacenamiento a la variable. Muchos autores prefieren mantener la diferencia entre ambos conceptos, ya que para otros identificadores la diferencia entre declaración y definición es importante.

2.11. DURACIÓN DE UNA VARIABLE

Dependiendo del lugar donde se definan las variables de C++, éstas se pueden utilizar en la totalidad del programa dentro de una función o pueden existir sólo temporalmente dentro de un bloque de una función. La zona de un programa en la que una variable está activa se denomina, normalmente, *ámbito* o *alcance* («*scope*»).

El *ámbito (alcance)* de una variable se extiende hasta los límites de la definición de su bloque. Los tipos básicos de variables en C++ son:

- variables *locales*;
- variables *globales*;
- variables *dinámicas*;
- variables *de campos de objetos*.

2.11.1. Variables locales

Las variables locales son aquéllas definidas en el interior de una función y son visibles sólo en esa función específica. Las reglas por las que se rigen las variables locales son:

1. En el interior de una función, una variable local no puede ser modificada por ninguna sentencia externa a la función.
2. Los nombres de las variables locales no han de ser únicos. Dos, tres o más funciones —por ejemplo— pueden definir variables de nombre Interruptor. Cada variable es distinta y pertenece a la función en que está declarada.
3. Las variables locales de las funciones no existen en memoria hasta que se ejecuta la función. Esta propiedad permite ahorrar memoria, ya que permite que varias funciones compartan la misma memoria para sus variables locales (pero no a la vez).

Por la razón dada en el punto 3, las variables locales se llaman también *automáticas* o *auto*, ya que dichas variables se crean automáticamente en la entrada a la función y se liberan también automáticamente cuando se termina la ejecución de la función.

```
#include <iostream>
using namespace std;

void main()
{
  int a, b, c, suma;   //variables locales
  int numero;          //variable local

  cout << "Cuántos números a sumar";
  cin >> número;

  suma = a + b + c;
}
```

2.11.2. Variables globales

Las *variables globales* son variables que se declaran fuera de la función y por defecto (omisión) son visibles a cualquier función incluyendo `main()`.

```
#include <iostream>

int a, b, c;   //declaración de variables globales

main()
{
  int valor;   //declaración de variable local
  //...
}
```

Todas las variables locales desaparecen cuando termina su bloque. Una variable global es visible desde el punto en que se define hasta el final del programa.

La memoria asignada a una variable global permanece a través de la ejecución del programa, tomando espacio válido según se utilice. Por esta razón, se debe evitar utilizar muchas variables globales dentro de un programa. Otro problema que surge con variables globales es que una función puede asignar un valor específico a una variable global. Posteriormente, en otra función, y por olvido, se pueden hacer cambios en la misma variable. Estos cambios dificultarán la localización de errores.

2.11.3. Variables dinámicas y de objetos

Las *variables dinámicas* o *punteros* tienen características que en algunos casos son similares tanto a variables locales como a globales. Al igual que una variable local, una variable dinámica se crea y libera durante la ejecución del programa. La *diferencia entre una variable local y una variable dinámica* es que la variable dinámica se crea tras su petición (en vez de automáticamente, como las variables locales), es decir, a su voluntad, y se libera cuando ya no se necesita. Al igual que una variable global, se pueden crear variables dinámicas que son accesibles desde múltiples funciones. Las variables dinámicas se examinan en detalle en el Capítulo 9 (*Punteros*).

Los objetos son tipos agregados de dato que pueden contener múltiples funciones y variables juntas en el mismo tipo de dato. En el Capítulo 13 se introducen los objetos, tipo de dato clave en C++ y sobre todo en programación orientada a objetos.

Q es una variable *global* por estar definida fuera de todos los bloques y es accesible desde todas las sentencias. Sin embargo, las definiciones dentro de `main`, como `A`, son *locales* a `main`. Por consiguiente, sólo las sentencias interiores a `main` pueden utilizar `A`.

```
# include <iostream.h>       Alcance o ámbito global
int Q;                       Q, variable global
void main()
{        ———————————————     Local a main
  int A;                     A, variable local

  A = 124;
  Q = 1;
  {      ———————————————     Primer subnivel en main
    int B;                   B, variable local

    B = 124;
    A = 457;
    Q = 2;
    {    ———————————————     Subnivel más interno de main
      int C;                 C, variable local

      C = 124;
      B = 457;
      A = 788;
      Q = 3;
    }
  }
}
```

2.12. SENTENCIA DE ASIGNACIÓN

A las variables se les puede dar un valor mediante el uso de una *sentencia de asignación*. Antes de que una variable sea utilizada, debe ser declarada.

Ejemplo

```
temperatura = 75.45;
cuenta = cuenta + 5;
espacio = velocidad * tiempo

int respuesta;     //resultado de una operación
respuesta = (1+4)*5;
```

Sintaxis

```
variable = expresión;   //ojo, = es asignación
                        //no igualdad
```

A la variable a la izquierda del signo igual se le asigna el valor de la expresión de la derecha; el punto y coma termina la sentencia.

Cuando se declara una variable se asigna almacenamiento para la variable y se pone un valor desconocido en su interior.

Ejemplo

```
#include <iostream>
using namespace std;

int main()
{
    aux = 4*5;
    cout << "Dos veces" << aux << "es:" << 2*aux << "\n";
    cout << "Tres veces" << aux << "ep:" << 3*aux << "\n";
    return (0)
}
```

2.13. ENTRADA/SALIDA POR CONSOLA

C++ no define directamente ninguna sentencia para realizar la entrada y salida de datos. La biblioteca de E/S proporciona un conjunto amplio de facilidades.

La biblioteca `iostream` contiene cuatro objetos de E/S: `cin`, `cout`, `cerr` y `clog`, que permiten la entrada y salida por consola.

Para utilizar esta biblioteca, su programa debe incluir las siguientes instrucciones al principio del archivo que contiene su programa:

```
#include <iostream>
using namespace std;
```

`iostream`, para manejar la entrada y salida, tiene dos tipos denominados `istream` y `ostream`, que representan flujos de entrada y salida, respectivamente. Un **flujo** (*stream*) es una secuencia de caracteres preparados para leer o escribir en un dispositivo de E/S de cualquier clase. El término «flujo» sugiere que los caracteres se generan o consumen secuencialmente en el tiempo.

Objetos estándar de entrada/salida

Para manejar la entrada, se utiliza un objeto del tipo `istream` llamado `cin`; este objeto se conoce también como la *entrada estándar*, que se conecta normalmente al teclado pero que se puede conectar a otro dispositivo. Para la salida, se utiliza un objeto `ostream` llamado `cout`, también conocido como *salida estándar*, que se conceta normalmente a la pantalla o monitor de la computadora, pero que se puede conectar a otro dispositivo.

La biblioteca define también otros dos objetos `ostream` denominados `cerr` y `clog`. El objeto `cerr` se conoce como el *error estándar* y se utiliza normalmente para generar mensajes de error y precaución (*warning*) a los usuarios de nuestros programas. El objeto `clog` se utiliza para información general sobre la ejecución del programa.

> ***Nota***
>
> En general, el sistema asocia cada uno de estos objetos con la ventana en la cual se ejecuta el programa. De este modo, cuando se leen datos de `cin`, los datos se leen de la ventana en la que se está ejecutando el programa y cuando escribimos con `cout`, `cerr` o `clog`, la salida se escribe en la misma ventana.

La mayoría de los sistemas operativos proporcionan un medio de redirigir los flujos de entrada o salida cuando se ejecuta un programa. Utilizando redirección se pueden asociar estos flujos con archivos de nuestra elección.

2.13.1. Entrada (`cin`)

El archivo `iostream` define `cin` como un objeto que representa el flujo. Para la salida, el operador << inserta caracteres en el flujo de salida. Para la entrada, `cin` utiliza el operador >> para extraer caracteres del flujo de entrada y por ello se denomina *operador de extracción*. Al igual que `cout`, es un objeto inteligente. Convierte la entrada, una serie de caracteres escritos desde el teclado, en un formato aceptable a la variable donde se almacena la información. Si no se redirige explícitamente `cin`, la entrada procede del teclado variable.



Figura 2.11. Entrada de una variable con `cin`.

La sentencia

```
cin >> numero
```

cuando se ejecuta espera a que el usuario teclee el valor de `numero`.

```
int n;      double x;
cin >> n;   cin >> x;
```

Al igual que <<, el operador >> se puede colocar en cascada.

```
cin >> variable1 >> variable2 >> ...;
```

Ejemplo

1. ```
   cout << "Introduzca número de pesos.\n";
        << "seguido del número de céntimos.\n";
   cin >> pesos >> centimos;
   ```
2. La sentencia `cin` también se puede escribir así:
   ```
   cin >> pesos
       >> centimos;
   ```

El archivo de cabecera `iostream.h` de la biblioteca C++ proporciona un flujo de entrada estándar `cin` y un *operador de extracción*, >>, para extraer valores del flujo y almacenarlos en variables. Si no se redirige explícitamente `cin`, la entrada procede del teclado.

```
int n;      double x;
cin >> n;   cin >> x;
```

Los operadores de extracción e inserción, >> y <<, apuntan en la dirección del flujo de datos.
```

El medio más sencillo para introducir respuestas por teclado es utilizar el flujo de entrada y el operador de extracción en unión de un flujo de salida y el operador de inserción. Un ejemplo típico es el siguiente:

```
cout << "Introduzca v1 y v2:";
cin  >> v1 >> v2 //lectura valores v1 y v2
cout << "v1= " << v1 << ",b= " << b << '\n';
cout << "Precio de venta al público";
cin  >> Precio_venta;
```

Ejemplo 2.6

*¿Cuál es la salida del siguiente programa, si se introducen por teclado las letras **LJ**?*

```
main()
{
    char primero, ultimo;
    cout << "Introduzca su primera y última inicial:";
    cin >> primero >> ultimo;
    cout << "Hola," << primero << "." << ultimo << ".!\n";
}
```

2.13.2. Salida (`cout`)

El *operador de inserción de flujo*, <<, inserta los datos a la derecha del mismo (operando derecho) en el objeto a la izquierda del operador (`cout`, operando izquierdo). Los valores de variables, así como cadenas de texto, se pueden sacar (visualizar) a la pantalla; también puede sacarse a la salida una combinación de variables y cadena. Por ejemplo:

```
cout << "Esto es una cadena"
```

visualiza

```
Esto es una cadena.
```

Es posible utilizar una serie de operadores << en cascada. Así:

```
cout << "Hola mundo C++" << "Soy Mackoy";
```

o bien

```
cout << NumeroDeAlumnos << "alumnos en clase de C++.";
```

La Figura 2.12 representa la salida con `cont`

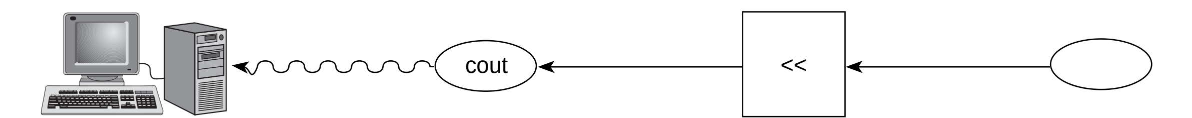


Figura 2.12. Salida de una cadena variable con `cout`.

cadena

```
cout << "Carchelejo y Cazalla"
```

operador de inserción

```
objeto cont
```

Ejemplo

Suponiendo que

```
i = 5, j = 12, C = 'A', n = 40.791512
```

la sentencia

```
cout << i << j << c << n;
```

visualiza en pantalla

```
512A40.791512
```

El manipulador nueva línea `\n`

C++ tiene un sistema fácil para indicar una nueva línea en la salida: la notación de '\n' del lenguaje C. Este carácter es un *carácter de escape* y cuando se encuentre en una cadena de caracteres, el siguiente carácter se combina con la barra inclinada (\) se forma una secuencia de escape denominado carácter «nueva línea». Aunque `\n` se escribe con dos símbolos, C++ considera `\n` como un único carácter (*nueva línea*). Si desea insertar una línea en blanco en la salida, escriba:

```
cout << "\n";
```

Considere la sentencia

```
cout << "Bienvenido\na\nC++!";
```

al ejecutarse se visualizará en pantalla el texto:

```
Bienvenido
a
C++
```

Ejemplo

```
cout << "Hola querido alumno.\n";
     << "Bienvenido a clase de C++\n";
```

produce dos líneas, una cadena en cada línea

```
Hola querido alumno
Bienvenido a clase de C++
```

El manipulador `endl`

Otro método para presentar una línea en blanco es utilizar el manipulador `endl`. En C++, `endl` es una notación especial que representa el concepto importante de comenzar una nueva línea.

```
cout << endl;   //produce una línea en blanco
```

Insertando `endl` en el flujo de salida hace que el cursor se mueva al principio de la siguiente línea.

Precaución

"\n\ y `endl` realizan la misma tarea, pero son ligeramente diferentes: `\n` se debe encerrar siempre entre comillas y `endl` no se escribe entre comillas.

`endl` está definida en el archivo de cabecera `iostream` y es parte del espacio de nombres `std`

```
cout << "\n"    //comienza una nueva línea
cout << endl;   //comienza una nueva línea
```

Obsérvese que la ejecución de `cout` no mueve automáticamente el cursor a la siguiente línea cuando se imprime una cadena, ya que la sentencia `cout` deja el cursor posicionado en el último carácter de la cadena de salida. La salida de cada sentencia `cout` comienza donde termina la última salida.

Ejemplo

```
cout << "El bueno, el";
cout << "feo,";
cout << "y el malo";
cout << endl;
```

visualiza la salida

```
El bueno, el feo, y el malo
```
— *cursor*

Escritura de nuevas líneas en pantalla

Para comenzar una nueva línea después de un texto de caracteres (cadena), se puede incluir `\n` al final de la cadena. Al ejecutar

```
cout << "Sierra Magina \n"
     << "está en Jaén, Andalucía \n";
```

se visualizará

```
Sierra Magina
está en Jaén, Andalucía
-
```

Consejo práctico

Es recomendable poner una instrucción de nueva línea al final de cada programa. Si el último elemento de una salida es una cadena, incluya `\n` al final de la cadena; en caso contrario, utilice `endl` como la última acción de salida de su programa.

2.13.3. Secuencias de escape

C++ utiliza *secuencias de escape* para visualizar caracteres que no están representados por símbolos tradicionales, tales como `\a`, `\b`, etc. Las secuencias de escape clásicas se muestran, de nuevo, en la Tabla 2.9.

Tabla 2.9. Caracteres secuencias de escape.

Secuencia de escape	Significado
`\a`	Alarma, suena el timbre (alarma) del sistema
`\b`	Retroceso de espacio
`\f`	Avance de página
`\n`	Nueva línea. Posiciona el cursor de la pantalla al principio de la siguiente línea
`\r`	Retorno de carro. Sitúa el cursor de la pantalla al principio de la línea actual; no avanza a la línea siguiente
`\t`	Tabulación horizontal. Mueve el cursor de la pantalla a la siguiente posición de tabulación
`\v`	Tabulación vertical
`\\`	Doble barra inclinada. Se utiliza para visualizar (imprimir) un carácter barra inclinada (*backslash*)
`\?`	Signo de interrogación
`\"`	Dobles comillas. Se utiliza para visualizar (imprimir) un carácter de dobles comillas
`\OOO`	Número octal
`\xhh`	Número hexadecimal
`\0`	Cero, nulo (ASCII 0)

Sentencia	Salida
`cout << "Bienvenido\n\a\C++";`	`Bienvenido` `a` `C++`
`cout << "Bienvenido a C++ \a\a\a";`	`Bienvenido a C++` `(seguido por tres pitidos)`
`cout << "Usar \\n para nueva línea";`	`Usar \n para nueva línea`
`cout << "\t Bienvenido a C++";`	`Bienvenido a C++`
`cout << "\"Texto entre comillas.\"";`	`"Texto entre comillas."`

Figura 2.12. Ejemplos de secuencias de escape.

Las secuencias de escape proporcionan flexibilidad en las aplicaciones mediante efectos especiales.

```
cout << "\n Error - Pulsar una tecla para continuar \n";

cout << "\n"          //salta a una nueva línea

cout << " Yo estoy preocupado\n no por el funcionamiento \n
          sino por la claridad .\n";
```

la última sentencia visualiza

```
Yo estoy preocupado
no por el funcionamiento

sino por la claridad.
```

debido a que la secuencia de escape `'\n'` significa *nueva línea* o *salto de línea*. Otros ejemplos pueden ser:

```
cout << "\n Tabla de números \n";   //uso de \n para nueva
                                    //línea

cout << "nNum1\t Num2\t Num3\n";    //uso de \t para
                                    //tabulaciones

cout << '\a';                       //uso de \a para alarma
                                    //sonora
```

en los que se utilizan los caracteres de secuencias de escape de *nueva línea* (`\n`), *tabulación* (`\t`) y *alarma* (`\a`).

Ejemplo 2.7

El listado `SECESC.CPP` *utiliza secuencias de escape, tales como emitir sonidos (pitidos) en el terminal dos veces y, a continuación, presentar dos retrocesos de espacios en blanco.*

```
// Programa:SECESC.CPP
// Autor J.R. Mortimer
// Propósito: Mostrar funcionamiento de secuencias de
// escape

#include <iostream>
using namespace std;

main()
{
  char sonidos='\a'; //secuencia de escape alarma en
                     //sonidos


  char bs='\b';      //almacena secuencia escape retroceso
                     //en bs

  cout << sonidos;   //envía secuencia de escape al
                     //terminal

  cout << sonidos;   //emite el sonido dos veces

  cout << "ZZ";      //imprime dos caracteres

  cout << bs;        //envía secuencia de escape al
                     //terminal

  cout << bs;        //mueve el cursor al primer carácter
                     //'Z'

  return 0;          //retorno de 0
}
```

Estilo de presentación

Si va a introducir datos desde la entrada (teclado) y desea visualizar un mensaje solicitando dichos datos y que éstos aparezcan en la misma línea del mensaje, no utilice el manipulador `\n` ni el `endl`.

Ejemplo

Cuando se ejecuta

```
cout << "Introduzca número de litros de aceite:";
cin >> numLitros
```

aparece en pantalla

```
Introduzca número de litros de aceite:
```

y cuando el usuario introduce, por teclado, la entrada, aparecerá en la misma línea (por ejemplo, si introduce `12.50`)

```
Introduzca número de litros de aceite: 12.50
```

Recuerde

En los nuevos compiladores ANSI/ISO C++ estándar, los programas que utilizan `cin` y `cont` para entrada y salida deben incluir el archivo `iostream` con las siguientes directivas:

```
#include <iostream>
using namespace std;
```

Si utiliza compiladores antiguos, no compatibles con el estándar ANSI/ISO C++, utilice entonces solamente:

```
#include <iostream.h>
```

De modo similar, otros nombres de bibliotecas son diferentes en compiladores antiguos. Este libro, normalmente, utiliza siempre el nuevo compilador estándar ANSI/ISO C++. Si utiliza un compilador antiguo, los nombres de los archivos de biblioteca terminan en `.h`.

TABLA 2.10. Archivos de cabecera nuevo estándar ANSI/ISO C++ *versus* antiguos compiladores

Nuevo archivo de cabecera	Archivo de cabecera antiguo equivalente
`cassert`	`assert.h`
`cctype`	`ctype.h`
`cstddef`	`stddef.h`
`cstdlib`	`stdlib.h`
`cmath`	`math.h`
`cstring`	`string.h`
`fstream`	`fstream.h`
`iomanip`	`iomanip.h`
`iostream`	`iostream.h`
`string`	`string` o *ninguna biblioteca se corresponde*

2.14. ESPACIO DE NOMBRES

Un **espacio de nombres** *(namespaces)* es una región declarativa con nombre opcional. El nombre de un espacio de nombres se puede utilizar para acceder a entidades declaradas en ese espacio de nombre; es decir, los miembros del espacio de nombre. En esencia, son conjuntos de variables, de funciones, de claves y de subespacios de nombre, miembros que siguen unas reglas de visibilidad. El espacio de nombres es una característica de C++ introducida en las últimas versiones, diseñada para simplificar la escritura de programas.

Un programa C++ puede dividirse en diferentes *espacios de nombres*. Un **espacio de nombre** es una parte del programa en el cual se recuerdan ciertos componentes y fuera de ese espacio son desconocidos o no son reconocidos.

Definición de un espacio de nombres

```
1.  namespace identificador {
           cuerpo_del_espacio_de nombre }  <——  no hay ;
2.  cuerpo_del_espacio_de_nombre
           sección_de_declaraciones   // miembros del espacio de nombre
```

Ejemplo

```
namespace geo
{
   const double PI = 3.141592;
   double longcircun (double radio)
      { return 2*PI*radio; }
}  //fin espacio de nombre geo
```

2.14.1. Acceso a los miembros de un espacio de nombres

El código externo a un espacio de nombre no puede acceder a los elementos que están en su interior por el método normal de una clase o una estructura. El espacio de nombre los convierte en invisibles.

Ejemplo de un mal uso

```
namespace geo
{
   const double PI = 3.141592;
   double longcircun (double radio)
      {return 2*PI*radio; }
{  //fin del espacio de nombre geo
double c = longcircun (16);   //error, no funciona
```

Para acceder al espacio de nombre se debe invocar al nombre del mismo cuando se refiera a ellos. Existen dos procedimientos para realizar este acceso.

Método 1. Preceder a cada nombre del elemento con el nombre del espacio de nombre y el operador de resolución de ámbito o alcance (::)

```
double c = geo::longcircuito (16);   //correcto
```

Método 2. Utilizar la directiva `using`

```
using namespace geo;
double c = longcircun (16);          //correcto
```

Regla

La directiva `using` produce que el espacio de nombre sea visible desde un punto hacia delante.

Sin embargo, se puede restringir la región en donde es efectiva la directiva `using` a un bloque específico.

```
void calculoSup()
{
   using namespace geo;
   //otro código fuente
   double c = longcircun (r);   //correcto
}
double c = longcircun (r);      //incorrecto
```

En este caso los miembros del espacio de nombre sólo son variables dentro del cuerpo de la función.

Ejemplo

```
int test;              //test global
namespace demo
{                      //comienzo de espacio demo
   int test;
}                      //fin de espacio demo

void main () {
   ::test = 0;         //test global
   demo::test = 10;    //test del espacio demo
}
```

Espacios de nombres en archivos de cabecera

Los espacios de nombre se utilizan con mucha frecuencia en los archivos de cabecera que contienen a clases o funciones de bibliotecas. Cada biblioteca tiene su propio espacio de nombre.

La biblioteca estándar de C++ contiene el espacio de nombres denominados `std`. Incluso después de incluir una cabecera como `<iostream>` es necesario invocar al flanco de salida `cont` como `std:cout`.

Ejemplo

```
std::cout << "Cada edad tiene sus costumbres.";
```

Para evitar la escritura de `std::` reiteradamente se utiliza la directiva `using`

```
using namespace std;
```

Esta sentencia significa que todas las sentencias que siguen están en el espacio de nombre `std`. `using` no es una declaración, sólo da acceso, en el espacio de nombres global, a aquellos nombres definidos en el mencionado espacio de nombres.

El estilo antiguo de los archivos de cabecera de C++, tal como `iostream.h` no utiliza espacios de nombres, pero el nuevo archivo de cabecera `iostream` debe utilizar el espacio de nombre `std`.

Ejemplo

```
#include <iostream>
using namespace std;   //utiliza todo el espacio de nombre std
```

Si no se utiliza la directiva `using` se deben preceder hasta los elementos con `std`

```
std::cout << "Hola mundo\h";
```

Sin embargo, es mucho más cómodo utilizar la directiva `using` y entonces se puede ejecutar

```
cout << "Hola mundo\n";
```

Espacios de nombres anidados

Los espacios de nombres se pueden anidar

```
namespace Exterior {
    int j;
    namespace Interior {
       void f()  { hj++; }   //Exterior::j
       int j;
       void g()  { j++; }   //Interior::j
    }
}
```

Ejemplo 2.8

Subespacios de nombre

```
namespace Continente
{
   namespace EstructurasDatos
   {
      class Lista ();
      class Pila ();
   }

   namespace FuncionesDatos
   {
      using namespace EstructuraDatos;
      void ordenarLista (Lista l);
   }
}
```

Una aplicación del espacio `Continente`

```
using namspace Continente;
void namespace Continente::EstructurasDatos

int main (int argc, char * argv[])
```
[4]

[4] La función `main` puede tener dos argumentos: número, `argc` y un puntero a `char` que contiene los nombres que recibe el programa, `char * argu[]`. Se estudiará más adelante.

```
{
   Lista l;
   return 0;
}
```

Espacios de nombres múltiples

Pueden existir varias instancias (ejemplares) de la lengua definición de espacios de nombres

```
namespace Geo
{
   cout double PI = 3.141592;
{ //fin del espacio de nombre Geo
// ...

namespace Geo
{
   double longcircun (double radio)
      { return 2 * PI * radio; }
}  //fin de espacio de nombre Geo
```

Permite una continuación de la misma definición y facilita que un espacio de nombre se utilice en diferentes archivos de cabecera, todos los cuales pueden incluirse en un archivo fuente.

Ejemplo 2.9

En la biblioteca estándar de C++, numerosos archivos de cabecera utilizan el espacio de nombres `std`

```
//archivoA.h
namespace alfa
{
   void funcA();
}

//archivoB.h
namespace alfa
{
   void funcB();
}
archivo Main.cpp
#include "archivoA.L"
#include "archivoB:L"
using namespace alfa;
funcA();
funcB();
```

Espacio de nombres sin nombre

Los espacios de nombres permiten mantener separados lógicamente grandes componentes de software como las bibliotecas, de modo que no se interfieran unas a otras.

Los nombres en C++ se pueden referir a variables, funciones. estructuras, enumeraciones, clases y miembros de estructuras y clases. Cuando los proyectos de programación van creciendo, el riesgo de que se produzca conflicto de nombres crece también. Si se utilizan bibliotecas de clase de más de una fuente, se pueden tener conflictos de nombres. Por ejemplo, dos bibliotecas pueden tener definidas clases tales `Lista`, `Arbol` y `Nodo`, pero on formatos incompatibles. Se puede desear la clase `Lista` de una biblioteca y la clase `Arbol` de la otra, y de cada una se puede esperar una versión propia de `Nodo`. Tales conflictos se resuelven con espacios de nombres calificados.

C++ estándar proporciona facilidades para proporcionar un mejor control sobre el ámbito o alcance de los nombres.

2.14.1. Aspectos prácticos

Un **espacio de nombres** (`namespace`) es una parte del programa en el que se reconocen ciertos nombres y fuera de su espacio son desconocidas. En la práctica es un conjunto o colección de definiciones de nombres.

Así, un nombre, tal como un nombre de función, puede tener definiciones diferentes en dos espacios de nombres distintos. Entonces, un programa puede utilizar uno de estos espacios de nombres en su lugar y otro en otra posición. Para utilizar cualquiera de las definiciones de `std` (de *standard*) en su programa debe insertar al principio la directiva `using`:

```
using namespace std;
```

de esta forma todas las sentencias de programa que vengan a continuación están dentro del espacio de nombres `std`. Clases, funciones y variables que antes eran un componente estándar de los compiladores C++, ahora están situados en el espacio de nombres `std`. Esto significa que las variables `cin`, `cout` y `endl` utilizadas para entradas y salidas en la consola (monitor) definidas en `iostream` se llaman realmente `std::cout`, `std::cin`, `std::endl`.

2.14.2. Reglas prácticas

1. Si quiere utilizar todos los nombres del espacio de nombres `std`, basta que incluya al comienzo de su programa

```
#include <iostream>
using namespace std;
```

2. Si desea utilizar solamente los nombres de `cin`, `cout` y `endl` de `std` (operaciones de la consola), debe comenzar con las siguientes líneas de programa (práctica utilizada por muchos programadores):

```
#include <iostream>
using std::cin;       //pone disponible cin
using std::cout;      //pone disponible cout
using std::endl;      //pone disponible endl
```

3. Si no desea utilizar el espacio de nombres `std` y reservar su uso para fines específicos, entonces utilice sólo

```
#include <iostream>
```

pero en este caso cuando invoque a las variables `cin`, `cout` o `endl`, deberá invocarlas siempre vinculadas a `std`; es decir, debe escribir sentencias tales como:

```
std::cout << "Hola mundo cruel.";
std::cout << std::endl;
```

Ejemplo

```
#include <iostream>

int main()
{
   std::cout << "Introduzca un valor";
   std::cin >> valor;
   std::cout << "El doble de " << valor << "es" valor * 2 << '\n';
   return(0);
{
```

Regla

Para evitar añadir `std::` decenas de veces en su programa, utilice la directiva `using namespace std;` o bien la individual como `using std::cout;`

2.14.3. La directiva `using` en programas multifunción

El espacio de nombres `std` es muy importante en la programación y existen diferentes filosofías de uso que cada programador deberá utilizar según sus necesidades y los criterios de diseño que utilice en sus programas.

Las diferentes selecciones para hacer disponibles los elementos del espacio de nombres `std` en un programa son:

1. Situar

   ```
   using namespace std;
   ```

   al principio de las definiciones de funciones de un archivo, haciendo disponibles todos los contenidos del espacio de nombres `std` a cada función del archivo.
2. Situar

   ```
   using namespace std;
   ```

   en una definición específica, haciendo todos los contenidos del espacio de nombres `std` disponibles a esa función específica.
3. En lugar de utilizar

   ```
   using namespace std
   ```

   se pueden situar directivas tales como

   ```
   using std::cin
   ```

   en una definición de función específica y hace que un elemento específico `cin` sólo esté disponible en esa función.
4. Se pueden omitir las directivas `using` y entonces se utiliza el prefijo `std::` siempre que se utilicen elementos del espacio de nombres `std`

   ```
   std::cout << "Uso de cout y endl del espacio de nombres std" << std::endl;
   ```

Ejemplo

```
#include <iostream>

int num;
int main()
{
   num = 3 * 6;
   std::cout << "Dos veces" << num << "es" << 2 * num << "\h";
   std::cout << "Tres veces" << num << "es" << 3 * num << "\h";
   return (0);
}
```

RESUMEN

Este capítulo le ha introducido a los componentes básicos de un programa C++. En posteriores capítulos se analizarán en profundidad cada uno de los componentes. Asimismo, aprenderá el modo de utilizar las características mejoradas en C++ que le permiten escribir programas orientados a objetos. En este capítulo ha aprendido lo siguiente:

- La estructura general de un programa C++.
- La mayoría de los programas C++ tienen una o más directivas `#include` al principio del programa fuente. Estas directivas `#include` proporcionan información adicional para crear su programa; éste, normalmente, utiliza `#include` para acceder a funciones definidas en archivos de biblioteca.
- Cada programa debe incluir una función llamada `main()`, aunque la mayoría de los programas tendrán muchas funciones además de `main()`.
- En C++ se utiliza el flujo `cin` y el operador de extracción >> para obtener entrada del teclado.
- Para visualizar salida a la pantalla, se utiliza el flujo `cout` y el operador de inserción <<.
- Los nombres de los identificadores deben comenzar con un carácter alfabético, seguido por un número de caracteres alfabéticos o numéricos, o bien, el carácter subrayado (_). El compilador normalmente ignora los caracteres posterior al 32.
- Los tipos de datos básicos de C++ son: enteros (`int`), entero largo (`long`), carácter (`char`), coma flotante (`float, double` y `long double`). Cada uno de los tipos enteros tiene un calificador `unsigned` para almacenar valores positivos. El tipo `double` tiene un tipo `long double` que incrementa el número de posiciones decimales significativos.
- Los tipos de datos carácter utilizan 1 byte de memoria; el tipo entero utiliza 2 bytes; el tipo entero largo utiliza 4 bytes y los tipos de coma flotante 4, 8 o 10 bytes de almacenamiento.
- Se utilizan conversiones forzosas de tipo o *moldes/ahormados* de tipos (*cast*) para convertir un tipo a otro. El compilador realiza automáticamente muchas conversiones de tipos. Por ejemplo, si se asigna un entero a una variable `float`, el compilador convierte automáticamente el valor entero a un tipo `float`.

Se puede seleccionar explícitamente una conversión de tipos precediendo la variable o expresión con (*tipo*), en donde tipo es un *tipo* de dato válido.

EJERCICIOS

2.1. ¿Cuál es la salida del siguiente programa?

```
#include <iostream.h>
main()
{
  // cout << "Hola mundo!\n";
}
```

2.2. ¿Qué es incorrecto en el siguiente programa?

```
#include <iostream.h>
// Este programa imprime "¡Hola mundo!":
main()
{
  cout << "Hola mundo!\n"
  return 0;
}
```

2.3. Escribir y ejecutar un programa que imprima su nombre y dirección.

2.4. Escribir y ejecutar un programa que imprima una página de texto con no más de 40 caracteres por línea.

2.5. Depurar el siguiente programa

```
// un programa C++ sin errores
#include <iostream.h>

void main()
{
  cout << "El lenguaje de programa-
           ción C++ << endl;
// { main()
{
```

2.6. ¿Cuál es la salida del siguiente programa, si se introducen por teclado J y M?

```
main()
{
  char primero, último;
  cout << "Introduzca sus inicia-
          les:\n";
  cout << "\t Primer apellido:";
  cin << primero;
  cout << "\t Segundo apellido:";
  cin << último;
  cout << "Hola, " << primero << ".
          " << último << ".!\n";
  return 0;
}
```

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 31).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

2.1. *¿Cuál es la salida del siguiente programa?*

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main()
{
    // cout << "Hola mundo!\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

2.2. *¿Cuál es la salida del siguiente programa?*

```
#include <cstdlib>
#include <iostream>
```

```
using namespace std;
#define prueba "esto es una prueba"
int main()
{
     char cadena[21]="sale la cade-
                      na.";
     cout <<prueba << endl;
     cout <<"Escribimos de
             nuevo.\n";
     cout << cadena << endl;
     cout << &cadena[8] << endl;
     system("PAUSE");
     return EXIT_SUCCESS;
}
```

2.3. *Escribir un programa que visualice la letra B con asteriscos.*

```
 *****
*     *
*     *
*     *
******
*     *
*     *
*     *
*****
```

2.4. *Codificar un programa en C++ que escriba en dos líneas distintas las frases:* `Bienvenido al C++; Pronto comenzaremos a programar en C.`

2.5. *Diseñar un programa en C que copie en un array de caracteres la frase «es un nuevo ejemplo en C++» y lo escriba en la pantalla.*

2.6. *¿Cuál es la salida del siguiente programa?*

```
#include <cstdlib>
#include <iostream>
#define Constante "de declaracion de
                   constante."
using namespace std;
int main( )
{
    char Salida[21]="Esto es un
                     ejemplo";
    cout << Salida << endl;
    cout << Constante << endl;
    cout << "Salta dos lineas\n \n";
    cout << "y tambien un\n";
    cout << &Salida[11];
    cout << " cadenas\n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

2.7. *¿Cuál es la salida del siguiente programa?*

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main( )
{
    char pax[] = "Juan Sin Miedo";
    cout << pax << "---->  "
            <<&pax[4] << endl;
    cout << pax << endl;
    cout << &pax[9] << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

2.8. *Escribir un programa que lea una variable entera y dos reales y lo visualice.*

2.9. *Escribir un programa que lea el largo y el ancho de un rectángulo.*

2.10. *¿Cuál de los siguientes identificadores son válidos?*

```
N              85 Nombre
MiProblema     AAAAAAAAAA
Mi Juego       Nombre_Apellidos
MiJuego        Saldo_Actual
write          92
m&m            Universidad Pontificia
registro       Set 15
* 143Edad
```

CAPÍTULO 3

Operadores y expresiones

Contenido

INTRODUCCIÓN

Los programas de computadoras se apoyan esencialmente en la realización de numerosas operaciones aritméticas y matemáticas de diferente complejidad. Este capítulo muestra cómo C++ hace uso de los operadores y expresiones para la resolución de operaciones. Los operadores fundamentales que se analizan en el capítulo son:

- aritméticos, lógicos y relacionales;
- de manipulación de bits;
- condicionales;
- especiales.

Además, se analizarán las conversiones de tipos de datos y las reglas que seguirá el compilador cuando concurran en una misma expresión diferentes tipos de operadores. Estas reglas se conocen como *prioridad* y *asociatividad.*

CONCEPTOS CLAVE

- Asignación.
- Asociatividad.
- Conversión explícita.
- Conversiones de tipos.
- Evaluación en cortocircuito.
- Expresión.
- Incrementación/decrementación.
- Manipulación de bits.
- Operador.
- Operador sizeof.
- Prioridad/precedencia.
- Tipo `bool`.

3.1. OPERADORES Y EXPRESIONES

Una *expresión* se compone de uno o más **operandos** que se combinan entre sí mediante **operadores**. En la práctica una expresión es una secuencia de operaciones y operandos que especifica un cálculo y en consecuencia devuelve un resultado. La forma más simple de una **expresión** consta de una única constante o variable. Las expresiones más complicadas se forman a partir de un operador y uno o más operandos.

Cada expresión produce un **resultado**. En el caso de una expresión sin operador/es el resultado es el propio operando; por ejemplo, una constante o una variable.

```
4.5           //expresión que devuelve el valor 2.5
Horas_Semana  //constante que devuelve un valor, p.e. 30
```

El resultado de expresiones que implican operadores se determina aplicando cada operador a sus operandos. Los **operadores** son símbolos que expresan operaciones que se aplican a uno o varios operandos y que devuelven un valor

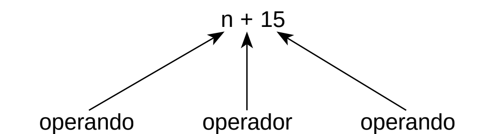


Figura 3.1. Expresión con un operador y operandos.

3.1.1. Expresiones

Al igual que en otros lenguajes, C++ permite formar expresiones utilizando variables constantes y operadores aritméticos: **+** (suma), **–** (resta), **×** (multiplicación, **/** (división) y **%** (resto módulos). Estas expresiones se pueden utilizar en cualquier lugar que sea legal utilizar un valor del tipo resultante de la expresión.

Una **expresión** es una operación que produce un valor con la excepción de expresiones `void`. Casi todos los tipos de sentencias utilizan expresiones de una u otra manera.

Ejemplo

Lista de expresiones, junto con la descripción del valor que cada una produce

```
x           // devuelve el valor de 15
25          // devuelve 25
x + 25      // devuelve x + 25
x = = 45    // comprueba la igualdad: devuelve 1 o 0
x = 50      // asignacion; devuelve el valor asignado (50)
i++         // devuelve el valor de i ante la incrementación
i = num +5  // expresión compleja; devuelve un nuevo
            // valor de i
```

Ejemplo

La declaración siguiente utiliza una expresión para su inicializador

```
int t = (100 + 50) /2;
```

Las expresiones más simples en C++ son simplemente literales o variables. Por ejemplo:

```
1.25   false   "sierra magina"   total
```

Expresiones más interesantes se forman combinando literales, variables y los valores de retorno de las funciones con diferentes operadores para producir menos valores. Las expresiones pueden contener a su vez expresiones como miembros o partes de ellas.

Ejemplo

Diferentes expresiones C++

```
i -> ObtenerValor(c) + 15
P * pow (1.0 + tasa, (double) mes))
new char[30]
sizeof (int) + sizeof (double) + 1
```

Nota

Obsérvese que las expresiones no son igual que las sentencias. Las sentencias indican al compilador que realice alguna tarea y termina con un punto y coma, mientras que las expresiones especifican un cálculo. En una sentencia puede haber varias expresiones.

3.1.2. Operadores

C/C++ son lenguajes muy ricos en operadores. Se clasifican en función del número de operandos sobre los que actúa y por las operaciones que realizan.

El significado de un operador —operación que realiza y tipo de resultado— depende de los tipos de sus operandos. Hasta que no se conoce el tipo de operando/s, no se puede conocer el significado de la expresión. Por ejemplo,

```
m + n
```

puede significar suma de enteros, concatenación de cadenas (`string`), suma de números en cuenta flotante, o incluso otra operación. En realidad, la expresión se evalúa y su resultado depende de los tipos de datos de `m` y `n`.

Los operadores se clasifican en: *unitarios* (*"unarios"*), binarios o ternarios. Operadores unitarios, tales como el operador de dirección (`&`) o el de *desreferencia* (*) que actúan sobre un operando. Operadores binarios, tales como suma (`+`) y resta (`-`) que actuan sobre dos operadores. Operadores ternarios, tales como, el operador condicional (`?:`) que actúa sobre tres operandos.

Algunos operadores pueden actuar como *"unitarios"* o como binarios; un ejemplo es el operador `*` que puede actuar como el operador de multiplicación (binario) o como operador de *desreferencia* (indirección). Los operadores imponen los requisitos sobre el tipo/s de su/s operando/s. Así, cuando un operador binario se aplica a operandos de tipos predefinidos o tipos compuestos, se requiere normalmente que sean del mismo tipo o tipos que se puedan convertir a un tipo común. Por ejemplo, se puede convertir un entero a un tipo de coma flotante y viceversa; sin embargo, no es posible convertir un tipo puntero a un tipo en coma flotante.

Los operadores se clasifican también según la posición del operador y de los operandos: *prefijo* (si va delante), *infijo* (si va en el interior) o *postfijo* (si va detras). Otra propiedad importante de los operadores es la **aridad** (número de operandos sobre los que actúa):

Unitarios (unarios o monarios)	= un solo operando
Binarios	= dos operandos
Ternarios	= tres operandos

Cuando las expresiones contienen varios operandos, otras propiedades son muy importantes: *precedencia*, *asociatividad* y *orden de evaluación* de los operandos.

Procedencia (*prioridad de evaluación*). Indica la prioridad del operador respecto a otros a la hora de calcular el valor de una expresión.

Asociatividad. Determina el orden en que se asocian los operandos del mismo tipo en ausencia de paréntesis.

Asociatividad por la derecha (D-I). Si dos operandos que actúan sobre el mismo operando tienen las misma precedencia se aplica primero al operador que está más a la derecha (operadores primarios, terciarios y asignación).

Asociatividad por la izquierda (I-D). En este caso se aplica primero al operador que está a mano izquierda (operadores binarios).

Ejemplos

`m = n = p`	*equivale a*	`m = (n = p)`	`D-I`
`m + n + p`	*equivale a*	`(m + n) + p`	`I-D`

Sin asociatividad. En algunos operadores no tiene sentido la asociatividad (éste es el caso de `sizeof`).

Ejemplo

```
20*5+24
```

Como se verá más adelante, el operador `*` tiene mayor prioridad que `+`.

1. Se evalúa primero `20*5`, produce el resultado `100`.
2. Se realiza la suma `100 + 24` y resulta `124`.

Una clasificación completa de operadores se muestra a continuación:

- Operadores de resolución de ámbito (alcance).
- Operadores aritméticos.
- Operadores de incremento y decremento.
- Operadores de asignación.
- Operadores de asignación compuesta.
- Operadores relacionales.
- Operadores lógicos.
- Operadores de bits.

- Operadores condicionales.
- Operadores de dirección o de indirección.
- Operadores de tamaño (`sizeof`).
- Operadores de secuencia o de evaluación (coma).
- Operadores de conversión.
- Operadores de *moldeado* o *molde* (`"cast"`).
- Operadores de construcción de tipos (`static_cast, reinterpret_cast, const_cast, dynamic_cast`).
- Operadores de memoria dinámica (`new` y `delete`).

Resumen

La *prioridad* o *precedencia* de operadores determina el orden en el que se aplican los operadores a un valor. Los operadores C++ vienen en una tabla con 16 grupos. Los operadores del grupo 1 tienen mayor prioridad que los del grupo 2, y así sucesivamente:

- Si dos operadores se aplican al mismo operando, el operador con mayor prioridad se aplica primero.
- Todos los operadores del mismo grupo tienen igual prioridad y asociatividad.
- Si dos operandos tienen igual prioridad, el operador con prioridad más alta se aplica primero.
- La asociatividad izquierda-derecha significa aplicar el operador más a la izquierda primero, y en la asociatividad derecha-izquierda se aplica primero el operador más a la derecha.
- Los paréntesis tienen la máxima prioridad.

<table>
<tr><th>Prioridad</th><th>Operadores</th><th>Asociatividad</th></tr>
<tr><td>1</td><td>:: × -> [] ()</td><td>I – D</td></tr>
<tr><td>2</td><td>++ –– ~ ! - + & * sizeof</td><td>D – I</td></tr>
<tr><td>3</td><td>.* ->*</td><td>I – D</td></tr>
<tr><td>4</td><td>* / %</td><td>I – D</td></tr>
<tr><td>5</td><td>+ –</td><td>I – D</td></tr>
<tr><td>6</td><td><< >></td><td>I – D</td></tr>
<tr><td>7</td><td>< <= > >=</td><td>I – D</td></tr>
<tr><td>8</td><td>== !=</td><td>I – D</td></tr>
<tr><td>9</td><td>&</td><td>I – D</td></tr>
<tr><td>10</td><td>^</td><td>I – D</td></tr>
<tr><td>11</td><td>||</td><td>I – D</td></tr>
<tr><td>12</td><td>&&</td><td>I – D</td></tr>
<tr><td>13</td><td>||</td><td>I – D</td></tr>
<tr><td>14</td><td>?: (expresión condicional)</td><td>D – I</td></tr>
<tr><td>15</td><td>= *= /= %= += -=
<<= >>= &= ||= ^=</td><td>D – I</td></tr>
<tr><td>16</td><td>, (operador coma)</td><td>I – D</td></tr>
<tr><td colspan="3">I - D : Izquierda – Derecha.
D - I : Derecha – Izquierda.</td></tr>
</table>

3.1.3. Evaluación de expresiones compuestas

Una expresión con dos o más operadores es una *expresión compuesta*. En una expresión compuesta el modo en que se agrupan los operandos a los operadores puede determinar el resultado de la expresión global. Dependiendo del modo en que agrupen los operandos el resultado será diferente.

La procedencia y asociatividad determina cómo se agrupan los operandos, aunque los programadores pueden anular estas reglas poniendo paréntesis en las expresiones particulares que seleccione.

Precedencia (prioridad)

Cada operador tiene una *precedencia* (prioridad). C++ utiliza las reglas de precedencia para decidir qué operador se utiliza primero. Los operadores con precedencia más alta se agrupan de modo que se evalúan lógicamente antes de los operadores con menor precedencia. La precerencia determina cómo analiza el compilador la expresión, no necesariamente el orden real de cálculo. Por ejemplo, en la expresión `x() + b() * c()`, la multiplicación tiene la precedencia más alta, pero `x()` puede ser llamado primero.

Ejemplo

¿Cual es el resultado de `6 + 3 * 4 / 2 + 2;`?

Dependiendo de como se agrupen las expresiones se producirá un resultado u otro. En concreto posibles soluciones son `14` (resultado real en C++), `36, 20` o `9`.

```
6 + 3 * 4 / 2 + 2;
     12
6 +     6     + 2
```
resultado en C++ `14`

El resultado `14` se produce porque la multiplicación (y la división) tiene la precedencia más alta y cuando coinciden en la misma expresión se aplica primero el operador de la izquierda, ya que los operadores aritméticos tienen asociatividad a izquierda. En el ejemplo anterior se realiza primero `3 * 4` y luego su valor se divide por `2`.

Anulación de la precedencia con paréntesis

Se puede anular la precedencia con paréntisis para tratar cada expresión entre paréntesis como una unidad y luego se aplican las reglas normales de precedencia dentro de cada expresión entre parántisis.

Ejemplo

```
cout << 6 + 3 * 4 / 2 + 2  << endl;            //visualiza 14
cout << ((6 + ((3 * 4)) / 2)) + 2)) << endl;   //visualiza 14
cout << 6 + 3 * 4 / (2 + 2)         << endl;   //visualiza 9
cout << (6 + 3) * (4 / 2 + 2)       << endl;   //visualiza 36
```

Asociatividad

Algunos operadores se agrupan de izquierda a derecha y otros operadores se agrupan de derecha a izquierda.

El orden de agrupamiento se denomina *asociatividad* del operador. Cuando dos operadores tienen la misma prioridad, C++ examina a ver si sus operadores tienen asociatividad *izquierda-a-derecha* o *derecha-a-izquierda*. La asociatividad izquierda-derecha significa que si dos operadores actúan sobre el mismo operando y tienen la misma precedencia, se aplica primero el operador situado a mano izquierda; en el caso de asociatividad a derecha se aplica primero el operador situado a mano derecha.

Ejemplo

- `x / y / z` *equivale a* `(x / y)/z`
- `x = y = z` *equivale a* `x =(y = z)`

```
float logos = 120/4*5;   //el valor es 150
```

3.2. OPERADOR DE ASIGNACIÓN

El operador de asignación = asigna el valor de la expresión derecha a la variable situada a su izquierda.

```
variable operador = expresión
```

`expresión` puede ser una variable, una constante o una expresión aritmética más complicada

```
codigo = 3467
fahrenheit = 123.456;
coordX = 525;
coordY = 725;
```

Asignación compuesta

Este operador es asociativo por la derecha, eso permite realizar asignaciones múltiples. Así,

```
a = b = c = 45;
```

equivale a

```
a = (b = (c = 45));
```

o dicho de otro modo, a las variables *a*, *b* y *c* se asigna el valor 45.

Esta propiedad permite inicializar varias variables con una sola sentencia

```
int a, b, c;
a = b = c = 5;    //se asigna 5 a las variables a, b y c
```

Ejemplo

```
int i, j, val_m;
const int ci = i;    //inicialización, no asignación
2040 = val_m;        //error
i + j = valor_m;     //error
ci = val_m;          //error
```

Además del operador de asignación =, C++ proporciona cinco operadores de asignación adicionales. En la Tabla 3.1 aparecen los seis operadores de asignación.

Tabla 3.1. Operadores de asignación de C++.

Símbolo	Uso	Descripción
`=`	`a = b`	Asigna el valor de *b* a *a*.
`*=`	`a *= b`	Multiplica a por *b* y asigna el resultado a la variable *a*.
`/=`	`a /= b`	Divide *a* entre *b* y asigna el resultado a la variable *a*.
`%=`	`a %= b`	Fija a al resto de *a*/*b*.
`+=`	`a += b`	Suma *b* y *a* y lo asigna a la variable *a*.
`-=`	`a -= b`	Resta *b* de *a* y asigna el resultado a la variable *a*.

Estos operadores de asignación actúan como una notación abreviada para expresiones utilizadas con frecuencia. Así, por ejemplo, si se desea multiplicar 10 por *i*, se puede escribir

```
i = i * 10;
```

Precedencia. Muchas expresiones inplican más de un operador. En estos casos es necesario saber qué operando se aplica primero para obtener el valor final.

C++ proporciona un operador abreviado de asignación (`*=`), que realiza una asignación equivalente:

`i *= 10;` *equivale a* `i = i * 10;`

Tabla 3.2. Equivalencia de operadores de asignación.

Operador	Sentencia abreviada	Sentencia no abreviada
`+=`	`m += n`	`m = m + n;`
`-=`	`m -= n`	`m = m - n;`
`*=`	`m *= n`	`m = m * n;`
`/=`	`m /= n`	`m = m / n;`
`%=`	`m %= n`	`m = m % n;`

Estos operadores de asignación no siempre se utilizan, aunque algunos programadores C++ se acostumbran a su empleo por el ahorro de escritura que suponen.

Operadores de asignación compuesta

C++ proporciona operadores de asignación compuesta para cada uno de los operadores. La sintaxis general de un operador de asignación compuesta es

```
a op= b;
```

donde `op=` puede ser cualquiera de los siguientes diez operadores:

```
+=    -=    *=   /=   %=   //operadores aritméticos
<<=   >>=   &=   ^=   |=   //operadores de bits
```

En esencia, cada operador compuesto equivale a:

```
a = a op b;
```

b puede ser una expresión a op= *expresión* y entonces la operación equivale a

```
a = a op(expresión)
```

Ejemplo

```
                          Equivalente a
cuenta += 5;              cuenta = cuenta + 5;
total -= descuento;       total = total – descuento;
cambio %= 100;            cambio = cambio % 100;
cantidad *= c1 + c2;      cantidad = cantidad * (c1 + c2);
m -= 8;                   m = m - 8;
m *= 6;                   m = m * 6;
```

3.3. OPERADORES ARITMÉTICOS

Los operadores aritméticos sirven para realizar operaciones aritméticas básicas. Los operadores aritméticos C++ siguen las reglas algebraicas típicas de jerarquía o prioridad. Estas reglas especifican la precedencia de las operaciones aritméticas.

3.3.1. Precedencia

Considere la expresión

```
3 + 5 * 2
```

¿Cuál es el valor correcto, `16 =(8*2)` o `13 =(3+10)`? De acuerdo a las citadas reglas, la multiplicación se realiza antes que la suma. Por consiguiente, la expresión anterior equivale a:

```
3 + (5 * 2)
```

En C++ las expresiones interiores a paréntesis se evalúan primero; a continuación, se realizan los operadores unitarios, seguidos por los operadores de multiplicación, división, resto (módulo), suma y resta.

Tabla 3.3. Operadores aritméticos.

Operador	Tipos enteros	Tipos reales	Ejemplo
+	Suma	Suma	4 + 5
-	Resta	Resta	7 – 3
*	Producto	Producto	4 * 5
/	División entera: cociente	División en coma flotante	8 / 5
%	División entera: resto	División en coma flotante	12 % 5

Tabla 3.4. Precedencia de operadores aritméticos básicos.

Operador	Operación	Nivel de precedencia
+, -	+25, -6.745	1
*, /, %	5*5 es 25	2
	25/5 es 5	
	25%6 es 1	
+, -	2+3 es 5	3
	2-3 es -1	

Obsérvese que los operadores + y -, cuando se utilizan delante de un operador, actúan como operadores unitarios más y menos.

```
+75      // significa que es positivo
-154     // significa que es negativo
```

Ejemplo 3.1

1. *¿Cuál es el resultado de la expresión:* `6 + 2 * 3 - 4 / 2`?

```
6 + 2 * 3 - 4 / 2
6 +  6  -  4 / 2
6 +  6  -   2
  12    -   2
      10
```

2. *¿Cuál es el resultado de la expresión:* `5 *5 (5+(6-2)+1)`*?*

```
5 * (5 + (6 – 2) + 1)
5 * (5 +   4   + 1)
5     *   10
      50
```

3.3.2. Asociatividad

En una expresión tal como

```
3 * 4 + 5
```

el compilador realiza primero la multiplicación —por tener el operador `*` prioridad más alta— y luego la suma, por tanto produce `17`. Para forzar un orden en las operaciones se deben utilizar paréntesis

```
3 * (4 + 5)
```

produce `27`, ya que `4+5` se realiza en primer lugar.

La *asociatividad* determina el orden en que se agrupan los operadores de igual prioridad; es decir, de izquierda a derecha o de derecha a izquierda. Por ejemplo,

`10 - 5 + 3` *se agrupa como* `(10 - 5) + 3`

ya que `–` y `+`, que tienen igual prioridad, tienen asociatividad de izquierda a derecha. Sin embargo,

```
x = y = z
```

se agrupa como

```
x = (y = z)
```

ya que su asociatividad es de derecha a izquierda.

Tabla 3.5. Prioridad y asociatividad.

Prioridad (mayor a menor)	**Asociatividad**
`+, - (unitarios)`	izquierda-derecha (I-D)
`*, /, %`	izquierda-derecha (I-D)
`+, -`	izquierda-derecha (I-D)

Ejemplo 3.2

¿Cuál es el resultado de la expresión: `7 * 10 –5 % 3 * 4 + 9`*?*

Existen tres operadores de prioridad más alta (`*, % y *`)

```
70 – 5 % 3 * 4 + 9
```

La asociatividad es a izquierda, por consiguiente se ejecuta a continuación `%`

```
70 – 2 * 4 + 9
```

y la segunda multiplicación se realiza a continuación, produciendo

```
70 – 8 + 9
```

Las dos operaciones restantes son de igual prioridad y como la asociatividad es a izquierda, se realizará la resta primero y se obtiene el resultado

```
62 + 9
```

y por último se realiza la suma y se obtiene el resultado final de

```
71
```

3.3.2. Uso de paréntesis

Los paréntesis se pueden utilizar para cambiar el orden usual de evaluación de una expresión determinada por su prioridad y asociatividad. Las subexpresiones entre paréntesis se evalúan en primer lugar

según el modo estándar y los resultados se combinan para evaluar la expresión completa. Si los paréntesis están *anidados* —es decir, un conjunto de paréntesis contenido en otro— se ejecutan en primer lugar los paréntesis más internos.

Por ejemplo, considérese la expresión `(7 * (10 – 5) % 3) * 4 + 9`. La subexpresión `(10 – 5)` se evalúa primero, produciendo

```
(7 * 5 % 3) * 4 + 9
```

A continuación, se evalúa de izquierda a derecha la subexpresión `(7 * 5 % 3)`

```
(35 % 3) * 4 + 9
```

seguida de

```
2 * 4 + 9
```

Se realiza a continuación la multiplicación, obteniendo

```
8 + 9
```

y la suma produce el resultado final

```
17
```

Precaución

Se debe tener cuidado en la escritura de expresiones que contengan dos o más operaciones para asegurarse que se evalúan en el orden previsto. Incluso aunque no se requieran paréntesis, deben utilizarse para clarificar el orden concebido de evaluación y escribir expresiones complicadas en términos de expresiones más simples. Es importante, sin embargo, que los paréntesis estén equilibrados —cada paréntesis a la izquierda tiene un correspondiente paréntesis a la derecha que aparece posteriormente en la expresión— ya que si existen paréntesis desequilibrados se producirá un error de compilación.

```
((8 – 5) + 4 – (3 + 7 )
```

error de compilación, falta paréntesis final a la derecha

3.4. OPERADORES DE INCREMENTO Y DECREMENTO

De las muchas características de C++ heredadas de C, una de las más útiles son los operadores de incremento ++ y decremento `––`. Los operadores ++ y `––`, denominados de *incrementación* y *decrementación*, suman o restan 1 a su argumento, respectivamente, cada vez que se aplican a una variable.

Por consiguiente,

Tabla 3.6. Operadores de incrementación (++) y decrementación (––).

Incrementación	Decrementación
`++n`	`--n`
`n += 1`	`n -= 1`
`n = n + 1`	`n = n – 1`

```
a++
```

es igual que

```
a+1
```

Las sentencias

```
++n;
n++;
```

tienen el mismo efecto; así como

```
--n;
n--;
```

Sin embargo, cuando se utilizan como expresiones tales como

```
m = n++;
```

o bien,

```
cout << --n;
```

`++n` produce un valor que es mayor en uno que el de `n++`, y `--n` produce un valor que es menor en uno que el valor de `n--`.

```
int a = 1, b;
b = a++;                         //b vale 1 y a vale 2

int a = 1, b;
b = ++a;                         //b vale 2 y a vale 2
```

> Si los operadores `++` y `--` están de prefijos, la operación de incremento se efectúa antes que la operación de asignación; si los operadores `++` y `--` están de sufijos, la asignación se efectúa en primer lugar y la incrementación o decrementación a continuación.

Ejemplo

```
int i = 10:
int j;
...
j = i++;
```

La variable `j` vale `10`, ya que cuando aparece `++` después del operando (la variable `i`) el valor que se asigna a `j` es el valor de `i(10)` y luego posteriormente se incrementa a `i`, que toma el valor 11.

Ejemplo 3.3

Demostración del funcionamiento de los operadores de incremento/decremento.

```
#include <iostream>
using namespace std;
```

```
// Test de operadores ++ y --
main()
{
  int m = 45, n = 75;
  cout << " m = " << m << " n = " << n << endl;
  ++m;
  --n;
  cout << " m = " << m << ", n = " << n << endl;
  m++;
  n--;
  cout << "m=" << m <<", n=" << n << endl;
  return 0;
}
```

Ejecución

```
m = 45,   n = 75
m = 46,   n = 74
m = 46,   n = 73
```

Ejemplo 3.4

Diferencias entre operadores de preincremento y postincremento.

```
#include <iostream>
using namespace std;

// test de operadores incremento y decremento
int main()
{
   int m = 99, n;
   n = ++m;
   cout << "m = " << m << ", n = " << n << endl;
   n = m++;
   cout << "m = " << m << ", n = " << n << endl;
   cout << "m = " << m++ << endl;
   cout << "m = " << m  << endl;
   cout << "m = " << ++m << endl;
   return 0;
}
```

Ejecución

```
m = 100,  n = 100
m = 101,  n = 100
m = 101
m = 102
m = 103
```

Ejemplo 3.5

Orden de evaluación no predecible en expresiones.

```
#include <iostream>
using namespace std;

void main()
{
   int n = 5, t;
   t = ++n * --n;
   cout << "n= " << n << ", t = " << t << endl;
   cout << ++n << " " << ++n << " " << ++n << endl;
}
```

Ejecución

```
n = 5,  t = 25
6  7  8
```

Aunque parece que aparentemente el resultado de `t` será `30`, en realidad es `25`, debido a que en la asignación de `t`, `n` se incrementa a 6 y, a continuación, se decrementa a 5 antes de que se evalúe el operador producto, calculando `5 * 5`. Por último, las tres subexpresiones se evalúan de derecha a izquierda y como la asociatividad a izquierda del operador de salida << no es significativa, el resultado será `6 7 8`, al contrario de `8 7 6`, que es lo que parece que aparentemente se producirá.

3.5. OPERADORES RELACIONALES

ANSI C++ soporta el tipo `bool` que tiene dos literales `false y true`. Una expresión *booleana* es, por consiguiente, una secuencia de operandos y operadores que se combinan para producir uno de los valores `true y false`.

C++ no tiene tipos de datos lógicos o booleanos, como Pascal, para representar los valores verdadero *(true)* y falso *(false)*. En su lugar se utiliza el tipo `int` para este propósito, con el valor entero 0 que representa a falso y distinto de cero a verdadero.

falso	cero
verdadero	distinto de cero

Operadores tales como >= y == que comprueban una relación entre dos operandos se llaman operadores relacionales y se utilizan en expresiones de la forma

expresión$_1$ *operador_relacional expresión*$_2$

expresión$_1$ *expresión*$_2$	expresiones compatibles C++
operador_relacional	un operador de la tabla 3.7

Los operadores relacionales se usan normalmente en sentencias de selección (`if`) o de iteración (`while, for`), que sirven para comprobar una condición. Utilizando operadores relacionales se realizan

operaciones de igualdad, desigualdad y diferencias relativas. La Tabla 3.7 muestra los operadores relacionales que se pueden aplicar a operandos de cualquier tipo de dato estándar: `char`, `int`, `float`, `double`, etc.

Cuando se utilizan los operadores en una expresión, el operador relacional produce un 0, o un 1, dependiendo del resultado de la condición. 0 se devuelve para una condición *falsa*, y 1 se devuelve para una condición *verdadera*. Por ejemplo, si se escribe

```
c = 3 < 7;
```

la variable *c* se pone a `1`, dado que como `3` es menor que `7`, entonces la operación < devuelve un valor de `1`, que se asigna a *c*.

Precaución

Un error típico, incluso entre programadores experimentales, es confundir el operador de asignación (=) con el operador de igualdad (==).

Tabla 3.7. Operadores relacionales de C++.

Operador	**Significado**	**Ejemplo**
`==`	*Igual a*	`a == b`
`!=`	*No igual a*	`a != b`
`>`	*Mayor que*	`a > b`
`<`	*Menor que*	`a < b`
`>=`	*Mayor o igual que*	`a >= b`
`<=`	*Menor o igual que*	`a <= b`

Ejemplo

- Si `x`, `a`, `b` y `c` son de tipo `double`, `número` es `int` e `inicial` es de tipo `char`, las siguientes expresiones booleanas son válidas:

```
x < 5.75
b * b >= 5.0 * a * c
numero == 100
inicial != '5'
```

- En datos numéricos, los operadores relacionales se utilizan normalmente para comparar. Así, si

```
x = 3.1
```

la expresión

```
x < 7.5
```

produce el valor `true`. De modo similar si

```
numero = 27
```

la expresión

```
numero == 100
```

produce el valor *false*.

- Los caracteres se comparan utilizando los códigos numéricos. (Véase Apéndice D, código ASCII, en página web del libro).

`'A' < 'C'` es verdadera (*true*), ya que `A` es el código 65 y es menor que el código 67 de `C`.
`'a' < 'c'` es verdadera (*true*): `a` (código 97) y `b` (código 99)
`'b' < 'B'` es false (*false*) ya que `b` (código 98) no es menor que `B` (código 66).

Los operadores relacionales tienen menor prioridad que los operadores aritméticos, y asociatividad de izquierda a derecha. Por ejemplo,

`m + 5 <= 2 * n` *equivale a* `(m + 5) <= (2 * n)`

Los operadores relacionales permiten comparar dos valores. Así, por ejemplo, (`if` significa *si*, se verá en el Capítulo 4)

```
if (Nota_asignatura < 9)
```

comprueba si `Nota_asignatura` es menor que 9. En caso de desear comprobar si la variable y el número son iguales, entonces utilizar la expresión

```
if (Nota_asignatura == 9)
```

Si por el contrario, se desea comprobar si la variable y el número no son iguales, entonces utilice la expresión

```
if (Nota_asignatura != 9)
```

3.6. OPERADORES LÓGICOS

Además de los operadores matemáticos, C++ tiene también *operadores lógicos*. Estos operadores se utilizan con expresiones para devolver un valor verdadero (cualquier entero distinto de cero) o un valor falso (0). Los operadores lógicos se denominan también *operadores booleanos*, en honor de George Boole, creador del álgebra de Boole.

Los operadores lógicos de C++ son: **`not (!), and (&&)`** y **`or (||)`**. El operador lógico ! (**`not`**, *no*) produce *falso* si su operando es *verdadero* (distinto de cero) y viceversa. El operador lógico **`&&`** (`and`, *y*) produce verdadero *sólo* si ambos operandos son *verdadero* (no cero); si cualquiera de los operandos es falso produce *falso*. El operador lógico **`||`** (**`or`**, *o*) produce verdadero si cualquiera de los operandos es verdadero (distinto de cero) y produce falso sólo si ambos operandos son falsos. La Tabla 3.8 muestra los operadores lógicos de C++.

Tabla 3.8. Operadores lógicos.

Operador	**Operación lógica**	**Ejemplo**
Negación (`!`)	**No** lógica	`!(x >= y)`
Y disfunción lógica (`&&`)	`operando_1 && operando_2`	`m < n && i > j`
O disfunción lógica `||`	`operando_1 || operando_2`	`m = 5 || n != 10`

Tabla 3.9. Tabla de verdad del operador lógico NOT (!).

`Operando (a)`	`NOT a (!a)`
Verdadero (1)	Falso (0)
Falso (0)	Verdadero (1)

Tabla 3.10. Tabla de verdad del operador lógico AND (&&).

Operandos		
`A`	`B`	`a && b`
Verdadero (1)	Verdadero (1)	Verdadero (1)
Verdadero (1)	Falso (0)	Falso (0)
Falso (0)	Verdadero (1)	Falso (0)
Falso (0)	Falso (0)	Falso (0)

Los valores booleanos (`bool`) son `verdadero` y `falso`.
`true` y `false` son constantes predefinidas de tipo `bool`.

Tabla 3.11. Tabla de verdad del operador lógico OR(||).

Operandos		
`A`	`B`	`a || b`
Verdadero (1)	Verdadero (1)	Verdadero (1)
Verdadero (1)	Falso (0)	Verdadero (1)
Falso (0)	Verdadero (1)	Verdadero (1)
Falso (0)	Falso (0)	Falso (0)

Al igual que los operadores matemáticos, el valor de una expresión formada con operadores lógicos depende de: (*a*) el operador y (*b*) sus argumentos. Con operadores lógicos existen sólo dos valores posibles para expresiones: *verdadero* y *falso*. La forma más usual de mostrar los resultados de operaciones lógicas es mediante las denominadas *tablas de verdad*, que muestran cómo funcionan cada uno de los operadores lógicos (Tablas 3.9, 3.10 y 3.11).

Ejemplos

```
!(7 == 5)
(aNum > 5) && (Nombre == "Mortimer")
(bNum > 3) || (Nombre == "Mortimer")
```

Los operadores lógicos se utilizan en expresiones condicionales y mediante sentencias `if, while` o `for`, que se analizarán en capítulos posteriores. Así, por ejemplo, la sentencia if (*si la condición es verdadera/falsa...*) se utiliza para evaluar operadores lógicos.

1.
```
if ((a < b) && (c > d))
   {
      cout << "Los resultados no son válidos";
   }
```

Si la variable *a* es menor que *b* y, al mismo tiempo, *c* es mayor que *d*, entonces visualizar el mensaje: `Los resultados no son válidos`.

2.
```
if ((ventas > 50000) || (horas < 100))
   {
      prima = 100000;
   }
```

Si la variable `ventas` es mayor `50000` o bien la variable `horas` es menor que `100`, entonces asignar a la variable prima el valor `100000`.

3.
```
if (!(ventas < 2500))
   {
      prima = 12500;
   }
```

En este ejemplo, si `ventas` es mayor que o igual a `2500`, se inicializará `prima` al valor `12500`.

> El operador `!` tiene prioridad más alta que &&, que a su vez tiene mayor prioridad que `||`. La asociatividad es de izquierda a derecha.

La precedencia de los operadores es: los operadores matemáticos tienen precedencia sobre los operadores relacionales, y los operadores relacionales tienen precedencia sobre los operadores lógicos.

La siguiente sentencia:

```
if (ventas < sal_min * 3 && anios > 10 * iva)...
```

equivale a

```
if ((ventas < (sal_min * 3)) && (anios > (10 * iva)))...
```

3.6.1. Evaluación en cortocircuito

En C++ los operandos de la izquierda de `&&` y `||` se evalúan siempre en primer lugar; si el valor del operando de la izquierda determina de forma inequívoca el valor de la expresión, el operando derecho no se evalúa. Esto significa que si el operando de la izquierda de `&&` es falso o el de `||`es verdadero, el operando de la derecha no se evalúa. Esta propiedad se denomina *evaluación en cortocircuito* y se debe a que si `p` es falso, la condición `p && q` es falsa, con independencia del valor de `q,` y de este modo C++ no evalúa `q`. De modo similar, si `p` es verdadera, la condición `p||q` es verdadera, con independencia del valor de `q,` y C++ no evalúa a `q`.

Ejemplo 3.6

Supongamos que se evalúa la expresión

```
(x >= 0.0) && (sqr(x) >= 2)
```

Dado que en una operación lógica `Y (&&)` si el operando de la izquierda *(x >= 0.0)* es falso *(x es negativo)*, la expresión lógica se evalúa a falso y, en consecuencia, no es necesario evaluar el segundo operando. En el ejemplo anterior la expresión evita calcular la raíz cuadrada de números (`x`) negativos.

La evaluación en **cortocircuito** tiene dos beneficios importantes:

1. Una expresión *booleana* se puede utilizar para *guardar* una operación potencialmente insegura en una segunda expresión *booleana*.
2. Se puede ahorrar una considerable cantidad de tiempo en la evaluación de condiciones complejas.

Ejemplo 3.7

1. *Los beneficios anteriores se aprecian en la expresión booleana*

   ```
   (n != 0) && (x < 1.0 / n)
   ```

   ya que no se puede producir un error de división por cero al evaluar esta expresión, pues si `n` es `0`, entonces la primera expresión

   ```
   n != 0
   ```

   es falsa y la segunda expresión

   ```
   x <1.0 / n
   ```

   no se evalúa.

   De modo similar, tampoco se producirá un error de división por cero al evaluar la condición

   ```
   (n == 0) || (x >= 5.0 / n)
   ```

   ya que si `n` es `0`, la primera expresión

   ```
   n == 0
   ```

   es verdadera y entonces no se evalúa la segunda expresión

   ```
   (x >= 5.0 / n)
   ```

Aplicación

Dado el test condicional

```
if ((7 > 5) || (ventas < 30) && (30 != 30))...
```

C++ examina sólo la primera condición, (`7 > 5`), ya que como es verdadera, la operación lógica `||(o)` será verdadera, sea cual sea el valor de la expresión que le sigue.

Otro ejemplo es el siguiente:

```
if ((8 < 4) && (edad > 18) && (letra_inicial == 'Z'))...
```

En este caso, C++ examina la primera condición y su valor es falso; por consiguiente, sea cual sea el valor que sigue al operador `&&`, la expresión primitiva será falsa y toda la subexpresión a la derecha de (`8 < 4`) no se evalúa por C++.

Por último, en la sentencia

```
if ((10 > 4) || (num = 0)) ...
```

la sentencia `num = 0` nunca se ejecutará.

3.6.2. Asignaciones *booleanas* (lógicas)

Las sentencias de asignación se pueden escribir de modo que se puede dar un valor de tipo `bool` o una variable `bool`.

Ejemplo

`MayorDeEdad = true;`	*asigna el valor true MayorDeEdad*
`MayorDeEdad = (x == y);`	*asigna el valor de x == y a MayorDeEdad*
`MayorDeEdad`	*cuando x e y son iguales, Mayor DeEdad es true y si no false.*

Ejemplo 3.8

Las sentencias de asignación siguientes asignan valores a los dos tipos de variables `bool`, `rango` y `es_letra`. La variable `rango` es verdadera (`true`) si el valor de n está en el rango –100 a 100; la variable `es_letra` es verdadera si `car` es una letra mayúscula o minúscula

```
rango   = (n > -100) && (n < 100);
esletra = (('A' <= car) && (car <= 'Z'))||
          (('a' <= car) && (car <= 'Z'));
```

La expresión de *a* es `true` si `n` cumple las condiciones expresadas (`n` mayor de –100 y menor de 100); en caso contrario es `false`. La expresión *b* utiliza los operadores && y `||`. La primera subexpresión (antes de `||`) es `true` si `car` es una letra mayúscula; la segunda subexpresión (después de `||`) es `true` si `car` es una letra minúscula. En resumen, `esLetra` es verdadera (`true`) si `car` es una letra, y false en caso contrario.

3.6.3. Expresiones booleanas

La mayoría de las sentencias de bifurcación que se verán en los Capítulos 4 y 5 están controladas por expresiones booleanas o lógicas. Una *expresión booleana* es cualquier expresión que es verdadera o falsa. La forma más simple de una expresión booleana consta de dos expresiones, tales como números o variables que se comparan con uno de los operadores de comparación o relacionales. Observe que algunos de los operadores se escriben con dos símbolos, `==`, `!=`, `<=`, `>=`, etc. y también, que se utiliza un signo doble igual para el signo igual y que se utilizan dos símbolos `!=` para el signo no igual (Tabla 3.7).

Operador "and" &&

Se pueden combinar dos comparaciones utilizando el operador `"and"` que se escribe &&.

Ejemplo

La expresión `(2 < x)&&(x < 7)` es verdad si `x` es mayor que `2` y `x` es menor que `7`, es caso contrario es falsa

```
(exp_booleana_1) && (expresión_booleana_2)
```

Aplicación

```
if (nota > 0) && (nota < 10)
   cout << "la calificación está entre 0 y 10.\n";
else
   cout << "la calificación no está entre 0 y 10.\n";
```

si el valor de nota es mayor que 0 y menor de 10, se ejecuta la primera sentencia `cont`, en caso contrario se ejecuta la segunda sentencia `cout`.

Ejemplo

Combinación de dos operadores "or" (acentos ||) en C++

```
(y < 0) || (y > 12)
```

es verdad si `y` es menor que `0` o `y` es mayor que 12.

Ejemplo

Se puede negar cualquier expresión booleana utilizando el operador !. Si desea negar una expresión booleana, situe la expresión entre paréntesis y ponga el operador ! delante de ella.

`!(x<y)`	*significa*	`"x` *es no menor que* `y"`
`!(x<y)`	*es equivalente a*	`x >= y`

Regla

Si desea utilizar una cadena de inigualdades tales como x<z<y, será mejor utilice la siguiente expresión = (x<z) && (z<y).

Operador or

Se puede formar una expresión booleana «or» combinando dos expresiones booleanas y utilizando el operador `or (||)`.

```
(expresión_booleana_1)||(expresión_booleana_2)
```

Aplicación

```
if ((x == 1)||(x == y))
   cont << "x es 1 o x es igual a y.\n";
else
   cont << "x ni es 1 ni igual a y.\n";
```

3.7. OPERADORES DE MANIPULACIÓN DE BITS

Una de las razones por las que C y C++ se han hecho tan populares en computadoras personales es que el lenguaje ofrece muchos operadores de manipulación de bits a bajo nivel.

Los operadores de manipulación o tratamiento de bits *(bitwise)* ejecutan operaciones lógicas sobre cada uno de los bits de los operandos. Estas operaciones son comparables en eficiencia y en velocidad a sus equivalentes en lenguaje ensamblador.

Cada operador de manipulación de bits realiza una operación lógica bit a bit sobre datos internos. Los operadores de manipulación de bits se aplican sólo a variables y constantes `char, int` y `long`, y no a datos en coma flotante. Dado que los números binarios constan de 1,s y 0,s (denominados *bits*), estos 1 y 0 se manipulan para producir el resultado deseado para cada uno de los operadores.

Las siguientes tablas de verdad describen las acciones que realizan los diversos operadores sobre los diversos patrones de bit de un dato `int` (`char` o `long`).

Tabla 3.12. Operadores lógicos bit a bit.

<table>
<tr><th>Operador</th><th>Operación</th></tr>
<tr><td>&</td><td>Y (AND) lógica bit a bit</td></tr>
<tr><td>||</td><td>O (OR) lógica (inclusiva) bit a bit</td></tr>
<tr><td>^</td><td>O (XOR) lógica (exclusiva) bit a bit (OR exclusive, XOR)</td></tr>
<tr><td>~</td><td>Complemento a uno (inversión de todos los bits)</td></tr>
<tr><td><<</td><td>Desplazamiento de bits a izquierda</td></tr>
<tr><td>>></td><td>Desplazamiento de bits a derecha</td></tr>
</table>

<table>
<tr><th>A&B == C</th></tr>
<tr><td>0&0 == 0
0&1 == 0
1&0 == 0
1&1 == 1</td></tr>
</table>

<table>
<tr><th>A||B == C</th></tr>
<tr><td>0||0 == 0
0||1 == 1
1||0 == 1
1||1 == 1</td></tr>
</table>

<table>
<tr><th>A^B == C</th></tr>
<tr><td>0^0 == 0
0^1 == 1
1^0 == 1
1^1 == 0</td></tr>
</table>

<table>
<tr><th>A</th><th>~A</th></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td>O</td><td>1</td></tr>
</table>

Ejemplo

1. Si se aplica el operador & de manipulación de bits a los números 9 y 14, se obtiene un resultado de 8. La Figura 3.2 muestra cómo se realiza la operación.

2.
```
(&)0x3A6B          = 0011   1010    0101   1011
   0x00F0          = 0000   0000    1111   0000
                     ----------------------------
0x3A6B  &  0x00F0  = 0000   0000    0101   0000 = 0x0050
```

3.
```
(||) 152 0x0098    = 0000   0000    1001   1000
     5  0x0005     = 0000   0000    0000   0101
                     ----------------------------
     152 || 5      = 0000   0000    1001   1101 = 0x009d
```

4.
```
(^)    83          = 0101   0011
      204          = 1100   1100
                     -----------
    83^204         = 1001   1101 = 157
```

```
 9  decimal equivale a    1 0 0 1     binario
                          & & & &
14  decimal equivale a    1 1 1 0     binario
                          -------
                        = 1 0 0 0     binario
                        = 8           decimal
```

Figura 3.2. Operador & de manipulación de bits.

3.7.1. Operadores de asignación adicionales

Al igual que los operadores aritméticos, los operadores de asignación abreviados están disponibles también para operadores de manipulación de bits. Estos operadores se muestran en la Tabla 3.13.

Tabla 3.13. Operadores de asignación adicionales.

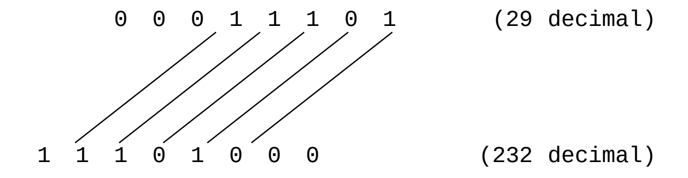
<table>
<tr><th>Símbolo</th><th>Uso</th><th>Descripción</th></tr>
<tr><td><=</td><td>a <<= b</td><td>Desplaza a a la izquierda b bits y asigna el resultado a a.</td></tr>
<tr><td>>>=</td><td>a >>= b</td><td>Desplaza a a la derecha b bits y asigna el resultado a a.</td></tr>
<tr><td>&=</td><td>a &= b</td><td>Asigna a a el valor a & b.</td></tr>
<tr><td>^=</td><td>a ^= b</td><td>Establece a a a ^ b.</td></tr>
<tr><td>|=</td><td>a | b</td><td>Establece a a a | b.</td></tr>
</table>

3.7.2. Operadores de desplazamiento de bits (>>, <<)

Equivalen a la instrucción `SHR (>>)` y `SHL (<<)` de los microprocesadores 80x86. Efectúa un desplazamiento a la derecha (>>) o a la izquierda (<<) de *n* posiciones de los bits del operando, siendo *n* un número entero. El número de bits desplazados depende del valor a la derecha del operador. Los formatos de los operadores de desplazamiento son:

1. *valor << numero_de_bits;*
2. *valor >> numero_de_bits;*

El *valor* puede ser una variable entera o carácter, o una constante. El *número_de_bits* determina cuántos bits se desplazarán. La Figura 3.2 muestra lo que sucede cuando el número 29 (binario 00011101) se desplaza a la izquierda tres bits con un desplazamiento a la izquierda bit a bit (<<).



Después de tres desplazamientos

Figura 3.2. Desplazamiento a la izquierda tres posiciones de los bits del número binario equivalente a 29.

Supongamos que la variable `num1` contiene el valor `25`, si se desplaza tres posiciones (`num << 3`), se obtiene el nuevo número `104 (11001000` en binario).

```
int num1 = 25;          //00011001 binario
int desp1, desp2;

desp1 = num1 << 3;      //11001000 binario
```

3.7.3. Operadores de direcciones

Son operadores que permiten manipular las direcciones de los objetos:

```
*expresión
&valor_i (lvalue)
objeto.miembro
puntero_hacia_objeto -> miembro
```

Tabla 3.14. Operadores de direcciones.

Operador	Acción
`*`	Lee o modifica el valor apuntado por la expresión. Se corresponde con un puntero y el resultado es del tipo apuntado.
`&`	Devuelve un puntero al objeto utilizado como operando, que debe ser un *lvalue* (variable dotada de una dirección de memoria). El resultado es un puntero de tipo idéntico al del operando.
`.`	Permite acceder a un miembro de un objeto agregado (unión, estructura o clase).
`->`	Accede a un miembro de un objeto agregado (unión, estructura o clase) apuntado por el operando de la izquierda.

3.8. OPERADOR CONDICIONAL

El operador condicional, `?:`, es un operador ternario que devuelve un resultado cuyo valor depende de la condición comprobada. Tiene asociatividad a derechas.

Al ser un operador ternario requiere tres operandos. El operador *condicional* se utiliza para reemplazar a la sentencia `if-else` lógica en algunas situaciones. El formato del operador condicional es:

```
expresión_c ? expresión_v : expresión_f;
```

Se evalúa *`expresión_c`* y su valor (cero = falso, distinto de cero = verdadero) determina cuál es la expresión a ejecutar; si la condición es verdadera se ejecuta *`expresión_v`* y si es falsa se ejecuta *`expresión_f`*.

La Figura 3.3 muestra el funcionamiento del operador condicional.

Otros ejemplos del uso del operador `?:` son:

```
n >= 0 ? 1 : -1   //1 si n es positivo, -1 si es negativo

m >= n ? m : n    //devuelve el mayor valor de m y n
```

```
(ventas > 150000)  ?  comisión = 100 : comisión = 0;
```

si ventas es mayor que 150.000 es	*si ventas no es mayor mayor que 150.000 ejecuta:*	*ejecutar*
`comision = 100`	`comision = 0`	

Figura 3.3. Formato de un operador condicional.

> La precedencia de `?` y `:` es menor que la de cualquier otro operando tratado hasta ese momento. Su asociatividad es de derechas.

3.9. OPERADOR COMA

El *operador coma* (`,`) de secuencia o evaluación permite combinar dos o más expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda y luego las restantes expresiones de izquierda a derecha. La expresión más a la derecha determina el resultado global. El uso del operador coma es como sigue:

```
expresión1, expresión2, expresión3, ..., expresiónn
```

Cada expresión se evalúa comenzando desde la izquierda y continuando hacia la derecha. Por ejemplo, en

```
int i = 10, j = 25;
```

dado que el operador coma se asocia de izquierda a derecha, la primera variable está declarada e inicializada antes que la segunda variable. Otros ejemplos son:

`i++, j++`	*equivale a*	`i++; j++;`
`i++, j++, k++`	*equivale a*	`i++; j++; k++;`

> El operador coma tiene prioridad de todos los operadores C++, y se asocia de izquierda a derecha.

El resultado de la expresión global se determina por el valor de *expresión*. Por ejemplo,

```
int i, j, resultado;
int i;
resultado = j = 10; i = j; i++;
```

El valor de esta expresión es `11`. En primer lugar, a `j` se asigna el valor `10`, a continuación a `i` se asigna el valor de `j`. Por último, `i` se incrementa a `11`.

La técnica del operador coma permite operaciones interesantes.

```
i = 10;
j = (i = 12, i + 8);
```

Cuando se ejecute la sección de código anterior, `j` vale `20`, ya que `i` vale `10` en la primera sentencia, en la segunda toma `i` el valor `12` y al sumar `i + 8` resulta `20`.

Ejemplos

1. `m = n++, p++;` *equivale a* `(m = n++), p++;`
2. Una aplicación tipica del operador coma se dá en la sentencia inicial de un bucle `for` (*véase* Capítulo 5).

   ```
   for (i=0, suma = 0,0; i<n; i++)
   ```

   primero se ejecuta la expresión `i=0`, luego la expresión `suma`

   ```
   suma = suma + i;
   ```

3.10. OPERADORES ESPECIALES `()`, `[]` Y `::`

3.10.1. El operador `()`

El operador `()` es el operador de llamada a funciones. Sirve para encerrar los argumentos de una función, efectuar conversiones explícitas de tipo, indicar en el seno de una declaración que un identificador corresponde a una función, resolver los conflictos de prioridad entre operadores.

3.10.2. El operador `[]`

Sirve para designar un elemento de un array. También se puede utilizar en unión con el operador **`delete`**; en este caso, indica el tamaño del array a destruir y su sintaxis es:

`delete` *`[tamaño_array] puntero_array;`*

Otros ejemplos son:

```
v[2]
return C[i - INFERIOR];
```

3.10.3. El operador `::`

Este operador es específico de C++ y se denomina *operador de ámbito de resolución*, y permite especificar el alcance o ámbito de un objeto. Su sintaxis es:

`class``::`*`miembro`* `o bien` *`::miembro`*

dato de referencia (variable, función, ...)

clase a la que pertenece el objeto

```
cout << i << "i <> :: i" << :: i << "\n";
void cuenta :: depósito (double amt)
{
   balance += amt;     //ojo, no está definido balance
}
```

3.11. EL OPERADOR SIZEOF

Con frecuencia, su programa necesita conocer el tamaño en bytes de un tipo de dato o variable. C++ proporciona el operador `sizeof`, que toma un argumento, bien un tipo de dato o bien el nombre de una variable (escalar, array, registro, etc.). El formato del operador es

```
sizeof(nombre_variable tipo_dato)
sizeof expresión
```

Ejemplo 3.9

Si se supone que el tipo `int` *consta de cuatro bytes y el tipo* `double` *consta de ocho bytes, las siguientes expresiones proporcionarán los valores 1, 4 y 8 respectivamente.*

```
sizeof (char)
sizeof (unsigned int)
sizeof (double).
```

El operador **`sizeof`** se puede aplicar también a expresiones. Así se puede escribir

```
cout << "La variable K es " << sizeof  k << " bytes";
cout << "La expresión a + b es " << sizeof (a + b) << " bytes"
```

El operador `sizeof` es un operador unitario, ya que opera sobre un valor único. Este operador produce un resultado que es el tamaño, en bytes, del dato o tipo de dato especificados. Debido a que la mayoría de los tipos de datos y variables requieren diferentes cantidades de almacenamiento interno en computadores diferentes, el operador `sizeof` permite consistencia de programas en diferentes tipos de computadores.

El operador `sizeof` se denomina también *operador en tiempo de compilación*, ya que, en tiempo de compilación, el compilador sustituye cada ocurrencia `de sizeof` en su programa con un valor entero sin signo (`unsigned`). El operador `sizeof` se utiliza en programación avanzada.

Ejercicio 3.1

Suponga que se desea conocer el tamaño, en bytes, de variables de coma flotante de su computadora. El siguiente programa realiza esta tarea:

```
// Nombre del archivo LONGBYTE.CPP
// Imprime el tamaño de valores de coma flotante

#include <iostream>
using namespace std;

int main()
{
   cout << "El tamaño de variables de coma flotante";
   cout << "de esta computadora es:" << sizeof(float)
           << '\n';
   return 0;
}
```

Este programa producirá diferentes resultados en diferentes clases de computadores. Compilando este programa bajo C++, el programa produce la salida siguiente:

```
El tamaño de variables de coma flotante de esta computadora es: 4
```

3.12. CONVERSIONES DE TIPOS

En C++ se puede convertir un valor de un tipo en un valor de otro tipo. Tal acción se denomina *conversión de tipos*. También se puede definir sus propios nombres de tipos utilizando la palabra reservada `typedef`. Esta característica es necesaria en numerosas expresiones aritméticas y asignaciones, debido a que las operadoras binarias requieren operandos del mismo tipo. Si no es este el caso, el tipo de un operando debe de ser convertido para coincidir con el otro operando. Cuando se llama a una función, el tipo de argumento debe coincidir con el tipo del parámetro; si no es así, el argumento se convierte de modo que coincidan sus tipos. C++ tiene operadores de conversión (`cant`) que le permiten definir una conversión explícitamente, o bien el compilador puede convertir automáticamente el tipo para su programa. C++ realiza muchas conversiones automáticamente:

- C++ convierte valores cuando se asigna un valor de un tipo aritmético a una variable de otro tipo aritmético.
- C++ convierte valores cuando se combinan tipos mezclados en expresiones.
- C++ convierte valores cuando se pasan argumentos a funciones.

En la práctica se realizan dos tipos de conversiones *implícitas* (ejecutadas automáticamente, algunas de las citadas anteriormente), o *explícitas* (solicitadas especialmente por el programador C++).

Conversión implícita

Los tipos fundamentales (básicos) pueden mezclarse libremente en asignaciones y expresiones. Las conversiones se ejecutan automáticamente: los operadores de tipo de precisión más baja (más pequeña) se convierten en tipos de más alta precisión. Esta conversión se denomina *promoción integral*.

Cuando se intenta realizar operaciones con tipos de datos de diferente precisión, el compilador define automáticamente —sin intervención del programador— de modo que convierte los operandos a un tipo común antes de realizar cualquier operación.

Por ejemplo, considere

```
int n = 0;
pi = 3.141592 + 3;
```

esta asignación a `pi`, de una suma de un entero y de uno real `(float)`, dos tipos diferentes, puede producir como valor de `pi`, `6`, o bien `6.141592` que sería lo lógico. El compilador realiza conversiones entre los tipos aritméticos que se definen para preservar, si es posible, la precisión. Lo normal sería que el entero se convierta en coma flotante y entonces se obtendrá el resultado `6.141592` que es de tipo `float (double)`. El siguiente paso que realizará el compilador será asignar el valor `double` a `n`, que es un entero. En el caso de la operación de asignación, prevalece el tipo del operando de la izquierda ya que no es posible cambiar el tipo del objeto que está en el lado izquierdo. Cuando los tipos izquierdo y derecho de una asignación difieren, el lado derecho se convierte al tipo del lado izquierdo. En el ejemplo anterior, si el valor real `(double)` se convierte a `int` el compilador producirá un mensaje de error similar a:

```
warning = assignment to 'int' from 'double'
```

que alertará al programador.

Reglas

1. En una expresión aritmética, los operadores binarios requieren operandos del mismo tipo. Si no coinciden, el tipo de un operando se debe convertir al tipo del otro operando.
2. En el caso de llamada a una función, el tipo de argumento debe corresponder con el tipo de parámetro; si no es así el argumento se convierte para que coincida su tipo.
3 C++ tiene operadores de *moldeado* de tipos (*moldes*, *«cast»*), que permiten definir una conversión de tipos, explícitamente, o bien el compilador puede convertir automáticamente el tipo si lo desea el programador.

Conversiones aritméticas

El lenguaje define un conjunto de conversiones entre los tipos incorporados. Entre éstas las más típicas son las *conversiones aritméticas* que aseguran que los operandos de un operador binario, tal como un operador aritmético o lógico, se convierten a un tipo común antes de que se evalúe el operador. Este tipo común es el tipo del resultado de la expresión. Las reglas definen una jerarquía de conversiones de tipos en la cual, los operandos, se convierten a los tipos de mayor precisión en las expresiones. Es decir, si un operando es del tipo `long double`, el otro operando también se convierte a `long double` sea cual sea su tipo.

La Tabla 3.15 considera el orden de los tipos de datos a efectos de conversión automática de tipos con indicación del tipo de dato de mayor precisión (el tamaño ocupado en bytes por el tipo de datos).

Tabla 3.15. Orden de los tipos de datos.

Tipo de dato	Precisión	Orden
`long double`	10 bytes	Más alta (19 dígitos)
`double`	8 bytes	15 dígitos
`float`	4 bytes	7 dígitos
`long`	4 bytes	No aplicable
`int`	4 bytes	No aplicable
`short`	2 bytes	No aplicable
`char`	1 bytes	Más baja
`bool`	No aplicable	

La *promoción de tipos* es una conversión de tipos aritméticos de modo que se convierte un tipo «más pequeño» a un tipo «mayor».

Por ejemplo, si un operando es de tipo `long double`, entonces el otro operando se convierte a `long double` independientemente del tipo que contiene. Cada uno de los tipos integrales más pequeños que `int` (`char`, `signed char`, `unsigned char`, `short` y `unsigned short`) se promueven a `int`, siempre que todos los valores posibles quepan en un `int`; en caso contrario se promueve a `unsigned int`. En el caso de los tipos lógicos (`bool`) se promueven a `int`, de modo que un valor *falso* se promueve a cero y un valor *verdadero* se promueve a uno.

El compilador ANSI/ISO C++ acepta definiciones de números con signo (`signed`) y sin signo (`unsigned`). En estos casos cuando un valor `unsigned` está implicado en una expresión, las reglas de conversión se definen para preservar el valor de los operandos, aunque en estos casos los tamaños relativos dependen de la máquina.

Ejemplo

```
bool    bandera;
char    car;
short   num_s;
int     num_i;
long    num_l;
float   num_f;
double  num_d;
long double   num_ld;
//algunas conversiones automáticas de tipos
num_d + num_i;     // se convierte a double
num_d + num_f;     // se convierte a double
num_i + num_d;     // se convierte a double
num_s + num_f;     // se convierte a int y luego a float
bandera = num_d;   // si num_d es 0, bandera es falso
car + num_f;       // car se convierte a int y después a float
```

Conversiones explícitas

Una *conversión de tipos* "*cast*" modelado de tipos (*moldes*) es una conversión explícita de tipos. C++ ofrece diferentes formas de modelar una expresión a un tipo diferente.

Notación compatible con C y versiones antiguas de C++

- *tipo* (*expr*)   //notación conversión de tipos, estilo función
- (*tipo*) *expr*;   //notación conversión estilo C

Notación compatible con estandar ANSI/ISO C++

C++ soporta los siguientes operadores de *molde*:

- *const_cast* <tipo> *(expr)*
- *dinamic_cast <tipo> (expr)*
- *reinterpret_cast* <tipo> *(expr)*
- *static_cast <tipo> (expr)*

Ejemplos

1. `(float)i     // convierte i a float`

2. `float (i)    // convierte i a float`

3.

```
varCar = static_cast <char> (varEntera):
```

tipo al que se convierte   *variable a convertir*

`varEntera` se convierte a `char` antes de ser asignada a `varCar`.

4. La conversión con C y compiladores antiguos de C++ de:

```
varCar = static_cast <char> (varEntera);
```

también se puede hacer con

```
varCar = (char) varEntera;
```

o alternativamente

```
varCar = char (varEntera);
```

Resumen

El compilador automáticamente realizará las conversiones de acuerdo a ciertas reglas. Para proteger la información, normalmente, sólo se permiten conversiones de menor tamaño (menos ricas) a mayor tamaño (más ricas) y no se pierde información; esta operación se llama *promoción*; por ejemplo, de `char` a `int` (no se pierde información). En otros casos se pierde información, como la conversión de `float` a `int`.

Si el compilador no puede efectuar la conversión por dudas en la misma, el programador puede hacerla explícita.

Ejemplo

```
int i = 7, j = 2;
double m = i/j;                  // truncamiento m == 3.0
m = (double) (i/j);              // m == (double) 3 == 3.0
m = (double) i/j;                // m == 3.5 i se convierte a double
m = i/(double) j;                // bien ahora j se convierte a double
m = (double) i/ (double) j;      // correcto
```

RESUMEN

Este capítulo examina los siguientes temas:

- Concepto de operadores y expresiones.
- Operadores de asignación: básicos y aritméticos.
- Operadores aritméticos, incluyendo +, -, *, / y % (módulos).
- Operadores de incremento y decremento. Estos operadores se aplican en formatos *pre* (anterior) y *post* (posterior). C++ permite aplicar estos operadores a variables que almacenan caracteres, enteros e incluso números en coma flotante.
- Operadores relacionales y lógicos que permiten construir expresiones lógicas. C++ no soporta un tipo lógico (*boolean*) predefinido y en su lugar considera 0 (cero) como *falso* y cualquier valor distinto de cero como *verdadero*.
- Operadores de manipulación de bits que realizan operaciones bit a bit (*bitwise*), `AND, OR, XOR` y `NOT`. C++ soporta los operadores de desplazamiento de bits `<< y >>`.
- Operadores de asignación de manipulación de bits que ofrecen formatos abreviados para sentencias simples de manipulación de bits.
- El operador coma, que es un operador muy especial, separa expresiones múltiples en las mismas sentencias y requiere que el programa evalúe totalmente una expresión antes de evaluar la siguiente.
- La expresión condicional, que ofrece una forma abreviada para la sentencia alternativa simple-doble `if-else`, que se estudiará en el Capítulo 4.
- Operadores especiales: `(), []` y ::.
- Conversión de tipos (*typecasting*) o moldeado, que permite forzar la conversión de tipos de una expresión.
- Reglas de prioridad y asociatividad de los diferentes operadores cuando se combinan en expresiones.
- El operador `sizeof`, que devuelve el tamaño en bytes de cualquier tipo de dato o una variable.

EJERCICIOS

3.1. ¿Cuál de los siguientes identificadores son válidos?

```
n              85 Nombre
MiProblema     AAAAAAAAAA
MiJuego        Nombre_Apellidos
Mi Juego       Saldo_Actual
write          92
m&m            Universidad Pontificia
registro       Set 15
A B            * 143Edad
```

3.2. Explicar la diferencia entre estas sentencias de asignación:

```
A-:= B       y        A-:= 'B'
A-:= 7       y        A-:= '7'
```

3.3. *X* es una variable entera e *Y* una variable carácter. ¿Qué resultados producirá la sentencia `cin << x << y;` si la entrada es

a) 5 c
b) 5C

3.4. Escribir un programa que lea un entero, lo multiplique por 2 y, a continuación, lo escriba de nuevo en la pantalla.

3.5. Escribir las sentencias de asignación que permitan intercambiar los contenidos (valores) de dos variables.

3.6. Escribir un programa que lea dos enteros en las variables *x* e *y* y, a continuación, obtenga los valores de. Ejecute el programa varias veces con diferentes pares de enteros como entrada.

3.7. Escribir un programa que solicite al usuario la longitud y anchura de una habitación y, a continuación, visualice su superficie con cuatro decimales.

3.8. Escribir un programa que convierta un número dado de segundos en el equivalente de minutos y segundos.

3.9. Escribir un programa que solicite dos números decimales y calcule su suma, visualizando la suma ajustada a la derecha. Por ejemplo, si los números son 57.45 y 425.55, el programa visualizará:

```
 57.45
425.55
483.00
```

3.10. ¿Cuáles son los resultados visualizados por el siguiente programa, si los datos proporcionados son 5 y 8?

```
#include <iostream.h>
#define M 6

void main (void)
{
  int A,B,C

  cin >> A >> B;
  C = 2 * A - B:
  C = C - M;
  B = A + C - M;
  A = B * M;
  Cout << A << end1
  B = -1
  cout << B << C
}
```

3.11. Escriba un programa para calcular la longitud de la circunferencia y el área del círculo para un radio introducido por el teclado.

3.12. Crear un programa para ver la diferencia entre los distintos formatos de salida.

3.13. Escribir un programa que visualice valores tales como:

```
7.1
7.12
7.123
7.1234
7.12345
7.123456
```

3.14. Escribir un programa que lea tres enteros y emita un mensaje que indique si están o no en orden numérico.

3.15. Escribir un programa que introduzca el número de un mes (1 a 12) y visualice el número de días de ese mes.

3.16. Se trata de escribir un programa que clasifique enteros leídos del teclado de acuerdo a los siguientes puntos: 1. Si 30 o mayor, o negativo, visualizar un mensaje en ese sentido; en caso contrario, si es un nuevo primo, potencia de 2, o un número compuesto, visualizar el mensaje correspondiente; si son cero o 1, visualizará 'cero' o 'unidad'.

3.17. Escribir un programa que lea dos números y visualice el mayor.

3.18. Codificar un programa que determine el mayor de tres números.

3.19. El domingo de Pascua es el primer domingo después de la primera luna llena posterior al equinoccio de primavera, y se determina mediante el siguiente cálculo sencillo:

```
A = anyo % 19
B = anyo % 4
C = anyo % 7
D = (19 * A + 24) % 30
E = (2 * B + 4 * C + 6 * D + 5) % 7
N = (22 + D + E)
```

donde *N* indica el número de día del mes de marzo (si *N* es igual o menor que 31) o abril (si es mayor que 31). Construir un programa que determine fechas de domingos de Pascua.

3.20. Codificar un programa que escriba la calificación correspondiente a una nota, de acuerdo con el siguiente criterio:

0 a < 5.0	Suspenso
5 a < 6.5	Aprobado
6.5 a 8.5	Notable
8.5 a 10	Sobresaliente
10	Matrícula de honor

3.21. Determinar si el carácter asociado a un código introducido por teclado corresponde a un carácter alfabético, dígito, de puntuación, especial o no imprimible.

PROBLEMAS

3.1. Escribir un programa que lea dos enteros de tres dígitos y calcule e imprima su producto, cociente y el resto cuando el primero se divide por el segundo. La salida será justificada a derecha:

```
  739        739
 x 12         12
 ----       ----
 8868      R = 7      Q = 61
```

3.2. Escribir un programa que dibuje el triángulo siguiente:

```
        *
      *   *
    *       *
  *           *
* * * * * * * *  *
```

3.3. Escribir un programa que dibuje el rectángulo siguiente:

```
* * * * * * * * * * *
*                   *
*                   *
*                   *
* * * * * * * * * * *
```

3.4. Modificar el programa anterior, de modo que se lea una palabra de cinco letras y se imprima en el centro del rectángulo.

3.5. Escribir un programa que escriba los números de 1 a 100 en una línea.

3.6. Escribir un programa en C que lea dos números y visualice el mayor.

3.7. Escribir un programa que lea dos enteros de tres dígitos e imprima su producto en el siguiente formato:

```
   325
 + 426
  1950
   650
  1300
------
138450
```

3.8. Teniendo como datos de entrada el radio y la altura de un cilindro queremos calcular: el área lateral y el volumen del cilindro.

3.9. Escribimos un programa en el que se introducen como datos de entrada la longitud del perímetro de un terreno, expresada con tres números enteros que representan hectómetros, decámetros y metros respectivamente. Se ha de escribir, con un rótulo representativo, la longitud en decímetros.

3.10. Construir un programa que calcule y escriba el producto, cociente entero y resto de dos números de tres cifras.

3.11. Codificar un programa que muestre un rectángulo en pantalla.

3.12. Construir un programa para obtener la hipotenusa y los ángulos agudos de un triángulo rectángulo a partir de las longitudes de los catetos.

3.13. Desglosar cierta cantidad de segundos introducida por teclado en su equivalente en semanas, días, horas, minutos y segundos.

3.14. Escribir un programa que le pregunte su nombre y le salude.

3.15. Escribir un programa que exprese cierta cantidad de euros en billetes y monedas de curso legal.

3.16. Cuatro enteros entre 0 y 100 representan las puntuaciones de un estudiante de un curso de informática. Escribir un programa para encontrar la media de estas puntuaciones y visualizar una tabla de notas de acuerdo al siguiente cuadro:

Media	*Puntuación*
90-100	A
80-89	B
70-79	C
60-69	D
0-59	E

3.17. La relación entre los lados (*a*, *b*) de un triángulo y la hipotenusa (*h*) viene dada por la fórmula

$$a^2 + b^2 = h^2$$

Escribir un programa que lea la longitud de los lados y calcule la hipotenusa.

3.18. El área de un triángulo cuyos lados son *a, b, c* se puede calcular por la fórmula

$$A = \sqrt{p\,(p - a)(p - b)(p - c)}$$

donde $p = (a + b + c)\,/2$. Escribir un programa que lea las longitudes de los tres lados de un triángulo y calcule el área del triángulo.

3.19. Escribir un programa que lea la hora de un día de notación de 24 horas y la respuesta en notación de 12 horas. Por ejemplo, si la entrada es 13:45, la salida será

```
1: 45 PM
```

El programa pedirá al usuario que introduzca exactamente cinco caracteres. Así, por ejemplo, las nueve en punto se introduce como

```
09:00
```

3.20. Escribir un programa que acepte fechas escritas de modo usual y las visualice como tres números. Por ejemplo, la entrada

```
15, Febrero 1989
```

producirá la salida

```
15 2 1989
```

3.21. Escribir un programa que acepte un número de tres dígitos escrito en palabra y, a continuación, los visualice como un valor de tipo entero. La entrada se termina con un punto. Por ejemplo, la entrada

```
doscientos veinticinco
```

producirá la salida

```
225
```

3.22. Escribir un programa que lea el radio de un círculo y, a continuación, visualice: circunferencia del círculo, área del círculo o diámetro del círculo.

3.23. Escribir un programa que acepte un año escrito en cifras arábigas y visualice el año escrito en números romanos, dentro del rango 1000 a 2000.

Nota: Recuerde que V = 5 X = 10 L = 50 C = 100 D = 500 M = 1000

```
V = 4         XL = 40     CM = 900
MCM = 1900   MCML = 1950  MCMLX = 1960  MCMXL = 1940
MCMLXXXIX = 1989
```

3.24. Un archivo de dados contiene los cuatro dígitos, A, B, C, D, de un entero positivo N: Se desea redondear N a la centena más próxima y visualizar la salida. Por ejemplo, si A es 2, B es 3, C es 6 y D es 2, entonces N será 2362 y el resultado redondeado será 2400. Si N es 2342, el resultado será 2300, y si N = 2962, entonces el número será 3000. Diseñar el programa correspondiente.

3.25. Un archivo contiene dos fechas en el formato día (1 a 31), mes (1 a 12) y año (entero de cuatro dígitos), correspondiente a la fecha de nacimiento y la fecha actual respectivamente. Escribir un programa que calcule y visualice la edad del individuo. Si es la fecha de un bebé (menos de un año de edad), la edad se debe dar en meses y días; en caso contrario, la edad se calculará en años.

3.26. Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4

(por ejemplo, 1984). Sin embargo, los años múltiplos de 100 sólo son bisiestos cuando a la vez son múltiples de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo será).

3.27. Escribir un programa que calcule el número de días de un mes, dados los valores numéricos del mes y el año.

3.28. Se desea calcular el salario neto semanal de los trabajadores de una empresa de acuerdo a las siguientes normas:

Horas semanales trabajadas < 38 a una tasa dada.
Horas extras (38 o más) a una tasa 50 por 100 superior a la ordinaria.
Impuestos 0 por 100, si el salario bruto es menor o igual a 300 euros.
Impuestos 10 por 100, si el salario bruto es mayor de 300 euros.

3.29. Construir un programa que indique si un número introducido por teclado es positivo, igual a cero, o negativo.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

3.1. Determinar el valor de las siguientes expresiones aritméticas:

15 / 12	15 % 12
24 / 12	24 % 12
123 / 100	200 % 100

3.2. ¿Cuál es el valor de cada una de las siguientes expresiones?

a) 10 * 14 – 3 * 2
b) –4 + 5 * 2
c) 13 – (24 + 2 * 5) / 4 %3
d) (4 – 40 / 5) %3
e) 4 * (3 + 5) – 8 * 4 % 2 – 5
f) –3 * 10 + 4 * (8 + 4 * 7 – 10 * 3) / 6

3.3. Escribir las siguientes expresiones aritméticas como expresiones de computadora: La potencia puede hacerse con la función pow(), por ejemplo, *(x + y)*^{2} *==pow(x+y,2).*

a) $\frac{x}{y} + 1$ *d*) $\frac{b}{c + d}$ *g*) $\frac{xy}{1 - 4x}$

b) $\frac{x + y}{x - y}$ *e*) $(a + b)\frac{c}{d}$ *h*) $\frac{xy}{mn}$

c) $x + \frac{y}{z}$ *f*) $[(a + b)^2]^2$ *i*) $(x + y)^2 . (a - b)$

3.4. Escribir las sentencias de asignación que permitan intercambiar los contenidos (valores) de dos variables *x*, e *y* de un cierto tipo de datos.

3.5. Escribir un programa que lea dos enteros en las variables *x* e *y*, y, a continuación, obtenga los valores de: *a*) *x* / y; *b*); *x % y*. Ejecute el programa varias veces con diferentes pares de enteros como entrada.

3.6. Una temperatura dada en grados Celsius (centígrados) puede ser convertida a una temperatura equivalente Fahrenheit de acuerdo a la siguiente fórmula: $f = \frac{9}{5} c + 32$. Escribir un programa que lea la temperatura en grados centígrados y la convierta a grados Fahrenheit.

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 49).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

3.1. La relación entre los lados (a,b) de un triángulo y la hipotenusa (h) viene dada por la fórmula: $a^2 + b^2 = h^2$. Escribir un programa que lea la longitud de los lados y calcule la hipotenusa.

3.2. Escribir un programa que lea un entero y, a continuación, visualice su doble y su triple.

3.3. Escriba un programa que lea los coeficientes *a, b, c , d, e, f* de un sistema lineal de dos ecuaciones con dos incógnitas y muestre la solución.

$$\begin{cases} ax + by = c \\ cx + dy = f \end{cases}$$

3.4. La fuerza de atracción entre dos masas, m_1 y m_2 separadas por una distancia d, está dada por la fórmula:

$$F = \frac{G\ .\ m1\ .\ m2}{d^2}$$

donde G es la constante de gravitación universal, $G = 6.673.x\ 10^{-8}\ cm^3/g.\ seg^2$
Escriba un programa que lea la masa de dos cuerpos y la distancia entre ellos y, a continuación, obtenga la fuerza gravitacional entre ella. La salida debe ser en dinas; un dina es igual a gr. cm/seg^2

3.5. La famosa ecuación de Einstein para conversión de una masa m en energía viene dada por la fórmula: $E = cm^3$, donde c es la velocidad de la luz y su valor es: $c = 2.997925 \partial\ 10^{10}$m/sg. Escribir un programa que lea una masa en gramos y obtenga la cantidad de energía producida cuando la masa se convierte en energía. Nota: Si la masa se da en gramos, la fórmula produce la energía en ergios.

3.6. Escribir un programa para convertir una medida dada en pies a sus equivalentes en: *a*) yardas; *b*) pulgadas; donde *c*) centímetros, y *d*) metros (1 pie = 12 pulgadas, 1 yarda = 3 pies, 1 pulgada = 2,54 cm, 1 m = 100 cm). Leer el número de pies e imprimir el número de yardas, pies, pulgadas, centímetros y metros.

3.7. Escribir un programa en el que se introduzca como datos de entrada la longitud del perímetro de un terreno, expresada con tres números enteros que representen hectómetros, decámetros y metros respectivamente, y visualice el perímetro en decímetros.

3.8. Escribir un programa que solicite al usuario una cantidad de euros y transforme la cantidad en euros en billetes y monedas de curso legal (cambio óptimo).

CAPÍTULO 4

Estructuras de selección: sentencias `if` y `switch`

Contenido

INTRODUCCIÓN

Los programas definidos hasta este punto se ejecutan de modo secuencial, es decir, una sentencia después de otra. La ejecución comienza con la primera sentencia de la función y prosigue hasta la última sentencia, cada una de las cuales se ejecuta una sola vez. Esta forma de programación es adecuada para resolver problemas sencillos. Sin embargo, para la resolución de problemas de tipo general se necesita la capacidad de controlar cuáles son las sentencias que se ejecutan y en qué momentos. Las *estructuras* o *construcciones de control* controlan la secuencia o flujo de ejecución de las sentencias. Las estructuras de control se dividen en tres grandes categorías en función del flujo de ejecución: *secuencia, selección* y *repetición.*

Este capítulo considera las *estructuras selectivas* o *condicionales* —sentencias `if` y `switch`— que controlan si una sentencia o lista de sentencias se ejecutan en función del cumplimiento o no de una condición. Para soportar estas construcciones, el estándar ANSI/ISO C++ soporta el tipo lógico `bool`.

CONCEPTOS CLAVE

- Estructura de control.
- Estructura de control selectiva.
- Sentencia `break`.
- Sentencia compuesta.
- Sentencia `enum`.
- Sentencia `if`.
- Sentencia `switch`.
- Tipo de dato `bool`.

4.1. ESTRUCTURAS DE CONTROL

Las **estructuras de control** controlan el flujo de ejecución de un programa o función. Las estructuras de control permiten combinar instrucciones o sentencias individuales en una simple unidad lógica con un punto de entrada y un punto de salida.

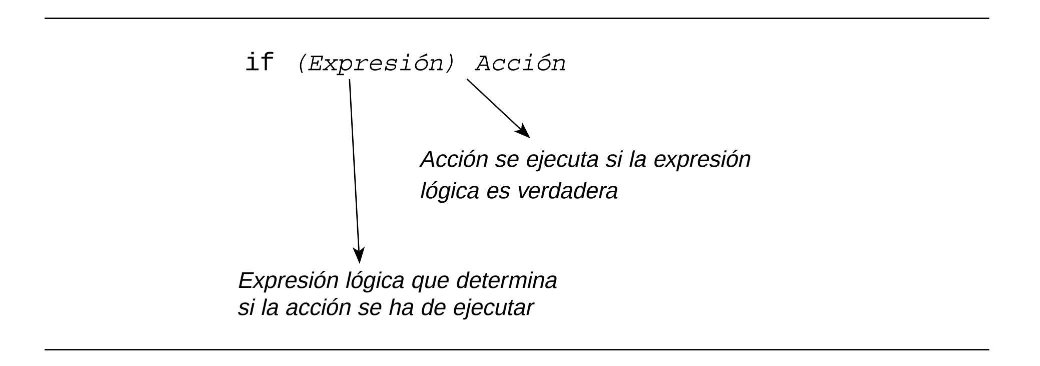
Las instrucciones o sentencias se organizan en tres tipos de estructuras de control que sirven para controlar el flujo de la ejecución: *secuencia*, *selección* (*decisión*) y *repetición*. Hasta este momento sólo se ha utilizado el flujo secuencial. Una **sentencia compuesta** es un conjunto de sentencias encerradas entre llaves ({ y }) que se utiliza para especificar un flujo secuencial.

```
{
  sentencia1;
  sentencia2;
  .
  .
  .
  sentencian;
}
```

El control fluye de la *sentencia*$_1$ a la *sentencia*$_2$, y así sucesivamente. Sin embargo, existen problemas que requieren etapas con dos o más opciones o alternativas a elegir en función del valor de una condición o expresión.

4.2. LA SENTENCIA `IF`

En C++, la estructura de control de selección principal es una sentencia `if`. La sentencia `if` tiene dos alternativas o formatos posibles. El formato más sencillo tiene la sintaxis siguiente:



La sentencia `if` funciona de la siguiente manera. Cuando se alcanza la sentencia `if` dentro de un programa, se evalúa la *expresión* entre paréntesis que viene a continuación de `if`. Si *Expresión* es verdadera, se ejecuta *`Acción`*; en caso contrario no se ejecuta *`Acción`* (en su formato más simple, *`Acción`* es una sentencia simple, y en los restantes formatos, es una sentencia compuesta). En cualquier caso la ejecución del programa continúa con la siguiente sentencia del programa. La Figura 4.1 muestra un *diagrama de flujo* que indica el flujo de ejecución del programa.

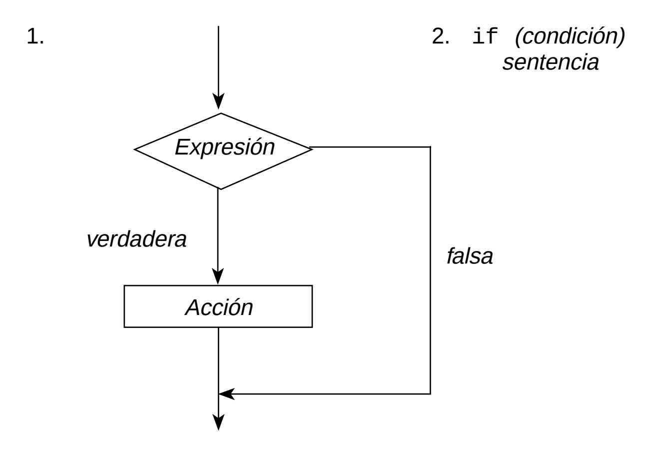


Figura 4.1. Diagrama de flujo de una sentencia básica `if`.

Otro sistema de representar la sentencia `if` es:

```
if (condición) sentencia;
```

`condición`	*es una expresión entera*
`sentencia`	*es cualquier sentencia ejecutable, que se ejecutará sólo si la condición toma un valor distinto de cero.*

Ejemplo 4.1

Prueba de divisibilidad.

```
void main()
{
    int n, d;
    cout << "Introduzca dos enteros:";
    cin >> n >> d;
    if (n%d == 0) cout << n << "es divisible por" << d << endl;
}
```

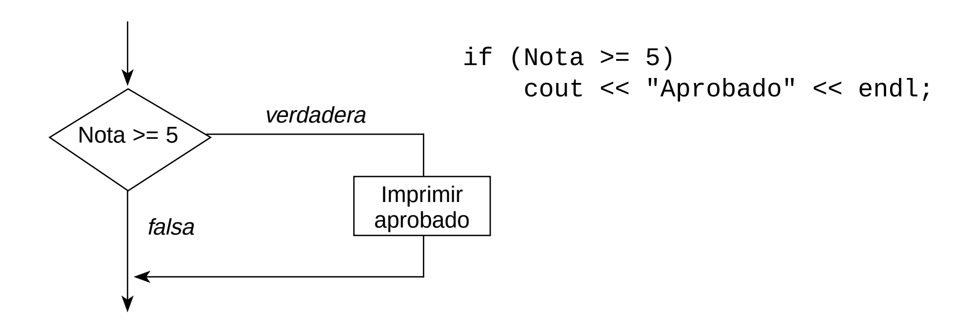
Ejecución

```
Introduzca dos enteros: 36 4
36 es divisible por 4
```

Este programa lee dos números enteros y comprueba cuál es el valor del resto de la división `n` entre `d` (`n % d`). Si el resto es cero, `n` es divisible por `d`. (En nuestro caso 36 es divisible por 4, ya que 36 : 4 = 9 y el resto es 0.)

Ejemplo 4.2

Representar la superación de un examen (Nota >= 5, Aprobado)



Ejemplo 4.3

```
// programa demo1 if
#include <iostream>      // E/S de C++
using namespace std;

void main()
{
   float numero;
   // obtener número introducido por usuario
   cout << "Introduzca un número positivo o negativo:";
   cin >> número;

   // comparar número con cero
   if (número > 0)
      cout << número << " es mayor que cero" << endl;
}
```

La ejecución de este programa produce

```
Introduzca un número positivo o negativo: 10.15
10.15 es mayor que cero
```

Si en lugar de introducir un número positivo se introduce un número negativo, ¿qué sucede? Nada. El programa es tan simple que sólo puede comprobar si el número es mayor que cero.

```
// programa demo2 if
#include <iostream>
using namespace std;

void main()
{
   float numero;

   // obtener numero introducido por usuario
   cout "introduzca un número positivo o negativo:";
   cin >> número;
   // comparar numero a cero
   if (numero > 0)
      cout << numero << "es mayor que cero" << endl;
   if (numero < 0)
      cout << numero << "es menor que cero" << endl;
```

```
   if (numero == 0)
      cout << numero << "es igual a cero" << endl;
}
```

Este programa simplemente añade otra sentencia `if` que comprueba si el número introducido es menor que cero. Realmente, una tercera sentencia `if` se añade también que, comprueba si el número es igual a cero.

Ejercicio 4.1

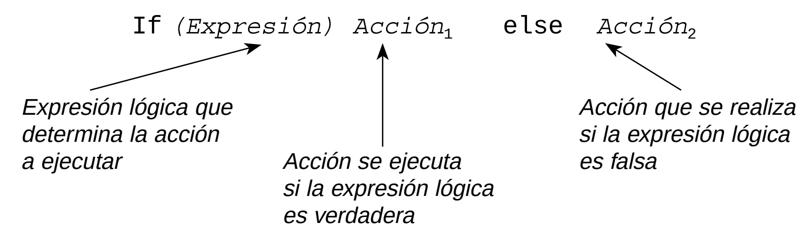
Visualizar el valor absoluto de un número leído del teclado.

```
// Programa de cálculo del valor absoluto de la entrada
#include <iostream>
#include <string>
using namespace std;

int main() {
   cout << "Introduzca un número:";
   int Valor;
   cin >> Valor;
   if (Valor < 0)
      Valor = -Valor;
   cout << Valor << "es positivo" << endl;
   return 0;
}
```

4.3. SENTENCIA: CONDICIÓN DOBLE `IF-ELSE`

Un segundo formato de la sentencia `if` es la sentencia `if-else`. Este formato de la sentencia `if` tiene la siguiente sintaxis:



En este formato *Acción*$_1$ y *Acción*$_2$ son individualmente o bien una única sentencia que termina en un punto y coma (;) o un grupo de sentencias encerrado entre llaves. Cuando se ejecuta la sentencia `if-else`, se evalúa *Expresión*. Si *Expresión* es verdadera, se ejecuta *Acción*$_1$ y en caso contrario se ejecuta *Acción*$_2$. La Figura 4.2 muestra la semántica de la sentencia `if-else`.

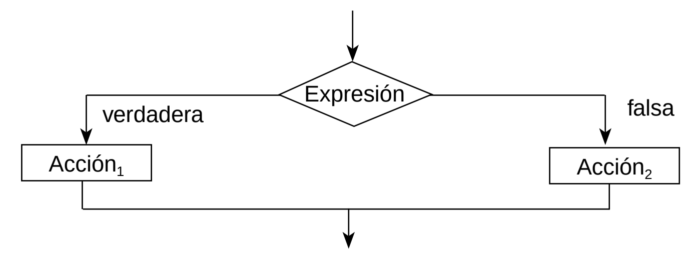


Figura 4.2. Diagrama de flujo de la representación de una sentencia `if-else`.

Ejemplos

1.
```
if (salario >= 100.000)
   salario_neto = salario – impuestos;
else
   salario_neto = salario;
```

Si `salario` es mayor que `100.000`, se calcula el salario neto, restándole los impuestos; en caso contrario (`else`), el salario neto es igual al salario (bruto).

2.
```
if (Nota >= 5)
    cout << "Aprobado" << endl;
else
  cout << "Suspenso" << endl;
```

Formatos

1.
```
if (expresión_lógica)
   sentencia
```

2.
```
if (expresión_lógica)
   sentencia1
else
   sentencia2
```

3.
```
if (expresión_lógica) sentencia
```

4.
```
if (expresión_lógica) sentencia1 else sentencia2
```

Si *expresión lógica* es verdadera, se ejecuta *`sentencia`* o bien *`sentencia`*$_1$, si es falsa (*si no*, en caso contrario), se ejecuta `sentencia`$_2$.

Ejemplos

```
1. if (x > 0.0)
      producto = producto * x;
2. if (x != 0.0)
      producto = producto * x;
   // se ejecuta la sentencia de asignación cuando x no es igual a 0.
   // en este caso producto se multiplica por x y el nuevo valor se
   // guarda en producto reemplazando el valor antiguo.
   // si x es igual a 0, la multiplicación no se ejecuta.
```

Ejemplo 4.4

Prueba de visibilidad (igual que 4.1, al que se ha añadido la cláusula `else`)

```
void main()
{
   int n, d;
   cout << "Introduzca dos enteros:";
```

```
    cin >> n >> d;
    if (n%d == 0) cout << n << "es divisible por" << d << endl;
    else
       cout << n << "no es divisible por" << d << endl;
}
```

Ejecución

```
Introduzca dos enteros 36 5
36 no es divisible por 5
```

Comentario

`36` no es divisible por `5`, ya que `36` dividido entre `5` produce un resto de 1 (`n % d = = 0`, es falsa, y se ejecuta la cláusula `else`).

Ejemplo 4.5

Calcular el mayor de dos números leídos del teclado y visualizarlo en pantalla.

```
void main()
{
   int x, y;
   cout << "Introduzca dos enteros:";
   cin >> x >> y;
   if (x > y) cout << x << endl;
   else
      cout << y << endl;
}
```

Ejecución

```
Introduzca dos enteros: 17 54
54
```

Comentario

La condición es (`x > y`). Si `x` es mayor que `y`, la condición es "verdadera" (*true*) y se evalúa a 1; en caso contrario la condición es "falsa" (*false*) y se evalúa a `0`. De este modo se imprime `x` cuando es mayor que `y`, como en el ejemplo de la ejecución.

4.4. SENTENCIAS `IF-ELSE` ANIDADAS

Hasta este momento, las sentencias `if` implementan decisiones que implican una o dos alternativas. En esta sección se mostrará cómo se puede utilizar la sentencia `if` para implementar decisiones que impliquen diferentes alternativas.

Una sentencia `if` es anidada cuando la sentencia de la rama verdadera o la rama falsa es a su vez una sentencia `if` . Una sentencia `if` anidada se puede utilizar para implementar decisiones con varias alternativas o multi-alternativas.

Sintaxis

```
if (condición1)
   sentencia1
else if (condición2)
   sentencia2
      .
      .
      .
else if (condiciónn)
   sentencian
else
   sentenciae
```

Ejemplo 4.6

```
// incrementar contadores de números positivos, números
// negativos o ceros
```

```
if (x > 0)
   num_pos = num_pos + 1;
else
   if (x < 0)
      num_neg = num_neg + 1;
   else
      num_ceros = num_ceros + 1;
```

La sentencia `if` anidada tiene tres alternativas. Se incrementa una de las tres variables (`num_pos`, `num_neg` y `num_ceros`) en 1, dependiendo de que `x` sea mayor que cero, menor que cero o igual a cero, respectivamente. Las cajas muestran la estructura lógica de la sentencia `if` anidada; la segunda sentencia `if` es la acción o tarea falsa (a continuación de `else`) de la primera sentencia `if`.

La ejecución de la sentencia `if` anidada se realiza como sigue: se comprueba la primera condición (`x > 0`); si es verdadera, `num_pos` se incrementa en 1 y se salta el resto de la sentencia `if`. Si la primera condición es falsa, se comprueba la segunda condición (`x < 0`); si es verdadera `num_neg` se incrementa en uno; en caso contrario se incrementa `num_ceros` en uno. Es importante considerar que la segunda condición se comprueba *sólo si* la primera condición es falsa.

4.4.1. Sangría en las sentencias `if` anidadas

El formato multibifurcación se compone de una serie de sentencias `if` anidadas, que se pueden escribir en cada línea una sentencia `if`. La sintasxis multibifurcación anidada es:

Formato 1

```
if (expresión_lógica1)
   sentencia1
```

```
else
   if (expresión_lógica2)
   else
      if (expresión_lógica3)
           sentencia3
         else
            if (expresión_lógica4)
              sentencia4
            else
               sentencia5
```

Formato 2

```
if (expresión_lógica1)
   sentencia1
else if (expresión_lógica2)
   sentencia2
else if (expresión_lógica3)
   sentencia3
else if (expresión_lógica4)
   sentencia4
else
   sentencia5
```

Ejemplos

1.
```
if (x > 0)
   if (y > 0)
      z = sqrt(x) + sqrt(y);
```

2.
```
if (x > 0)
   if (y > 0)
      z = sqrt(x) + sqrt(y);
else
   cerr << "\n *** Imposible calcular z" << endl;
```

Ejemplo 4.7

```
// comparación_if
// ilustra las sentencias compuestas if-else
#include <iostream>
using namespace std;

void main()
   cout << " introduzca un número positivo o negativo: ";
   cin >> número;

   // comparar número a cero
   if (numero > 0)
   {
      cout << numero << " es mayor que cero" << endl;
```

```
        cout << "pruebe de nuevo introduzca un número negativo" << endl;
     }
     else if (numero < 0)
        {
        cout << numero << " es menor que cero" << endl;
        cout << "pruebe de nuevo introduciendo un número negativo"
             << endl;
        }
        else
        {
           cout << numero << " es igual a cero" << endl;
           cout << "  ¿por qué no introduce un número negativo? << endl;
        }
     }
```

4.4.2. Comparación de sentencias `if` anidadas y secuencias de sentencias `if`

Los programadores tienen dos alternativas: (1) usar una secuencia de sentencias `if`; (2) una única sentencia `if` anidada. Por ejemplo, la sentencia `if` del Ejemplo 4.6 se puede reescribir como la siguiente secuencia de sentencias `if`.

```
if (x > 0)
   num_pos = num_pos + 1;
if (x < 0)
   num_neg = num_neg + 1;
if ( x == 0)
   num_ceros = num_ceros + 1;
```

Aunque la secuencia anterior es lógicamente equivalente a la original, no es tan legible ni eficiente. Al contrario que la sentencia `if` anidada, la secuencia no muestra claramente cuál es la sentencia a ejecutar para un valor determinado de `x`. Con respecto a la eficiencia, la sentencia `if` anidada se ejecuta más rápidamente cuando `x` es positivo ya que la primera condición (`x > 0`) es verdadera, lo que significa que la parte de la sentencia `if` a continuación del primer `else` se salta. En contraste, se comprueban siempre las tres condiciones en la secuencia de sentencias `if`. Si `x` es negativa, se comprueban dos condiciones en las sentencias `if` anidadas frente a las tres condiciones de las secuencias de sentencias `if`.

Una estructura típica `if-else` anidada permitida es:

```
if (número > o)
{
   // ...
}
else
   {
   if (// ...)
      {
         // ...
      }
   else
      {
```

```
      if (// ...)
         {
            // ...
         }
      }
   // ...
   }
```

Ejercicio 4.2

Existen diferentes formas de escribir sentencias `if` *anidadas.*

1.
```
if (a > 0) if (b > 0) ++a; else if (c > 0)
if (a < 5) ++b; else if (b < 5) ++c; else --a;
else if (c < 5) --b; else --c; else a = 0
```

2.
```
if (a > 0)                          // forma más legible
   if (b > 0) ++a;
   else
      if (c > 0)
         if (a < 5) ++b;
         else
            if (b < 5) ++c;
            else --a;
      else
         if (c < 5) --b;
         else --c;
else
   a = 0;
```

3.
```
if (a > 0)                          // forma más legible
   if (b > 0) ++a;
   else if (c > 0)
      if (a < 5) ++b;
      else if (b < 5) ++c;
      else --a;
   else if (c < 5) --b;
   else --c;
else
   a = 0
```

Ejercicio 4.3

Calcular el mayor de tres números enteros.

```
void main()
{
   int a, b, c, mayor;
   cout << "Introduzca tres enteros:";
   cin >> a >> b >> c;
```

```
   if (a > b)
      if (a > c) mayor = a;
      else mayor = c;
   else
      if (b > c) mayor = b;
      else mayor = c;
   cout << "El mayor es " << mayor << endl;
}
```

Ejecución

```
Introduzca tres enteros: 77 54 85
El mayor es 85
```

Análisis

Al ejecutar el primer `if`, la condición (`a > b`) es verdadera, entonces se ejecuta la segunda `if`. En el segundo `if` la condición (`a > c`) es falsa, en consecuencia el primer `else` y `mayor = 85` y se termina la sentencia `if` y se ejecuta la última línea y se visualiza `El mayor es 85`.

4.5. SENTENCIA `SWITCH`: CONDICIONES MÚLTIPLES

La sentencia `switch` es una sentencia C++ que se utiliza para seleccionar una de entre múltiples alternativas. La sentencia `switch` es especialmente útil cuando la selección se basa en el valor de una variable simple o de una expresión simple denominada *expresión de control* o *selector*. El valor de esta expresión puede ser de tipo `int` o `char`, pero no de tipo `double` .

Sintaxis

```
switch (selector)
{
   case etiqueta1 : sentencias1;
                    break;
   case etiqueta2 : sentencias2;
                    break;
      .
      .
      .
   case etiquetan : sentenciasn;
                    break;
   default:          sentenciasd;        // opcional
}
```

La expresión de control o *selector* se evalúa y se compara con cada una de las etiquetas de `case`. La expresión *selector* debe ser un tipo ordinal (por ejemplo, `int, char, bool` pero no `float o string`). Cada *etiqueta* es un valor único, constante, y cada etiqueta debe tener un valor diferente de los otros. Si el valor de la expresión selector es igual a una de las etiquetas `case` —por ejemplo, *etiqueta*$_i$— entonces la ejecución comenzará con la primera sentencia de la secuencia *secuencia*$_i$ y continuará hasta que se encuentra una sentencia `break` (o hasta que se encuentra el final de la sentencia de control `switch`).

El tipo de cada etiqueta debe ser el mismo que la expresión de *selector*. Las expresiones están permitidas como etiquetas pero sólo si cada operando de la expresión es por sí misma una constante —por ejemplo, `4 + 8` o bien `m * 15`, siempre que `m` hubiera sido definido anteriormente como constante con nombre.

Si el valor del selector no está listado en ninguna etiqueta `case`, no se ejecutará ninguna de las opciones a menos que se especifique una acción por defecto (omisión). La omisión de una etiqueta `default` puede crear un error lógico difícil de prever. Aunque la etiqueta `default` es opcional, se recomienda su uso a menos que se esté absolutamente seguro de que todos los valores de *selector* estén incluidos en las etiquetas `case`.

Una sentencia `break` consta de la palabra reservada `break` seguida por un punto y coma. Cuando la computadora ejecuta las sentencias siguientes a una etiqueta `case`, continúa hasta que se alcanza una sentencia `break`. Si la computadora encuentra una sentencia `break`, termina la sentencia `switch`. Si se omiten las sentencias `break`, después de ejecutar el código de `case`, la computadora ejecutará el código que sigue a la siguiente `case`.

Ejemplo 1

```
switch (opción)
{
   case 0:
      cout << "Cero!" << endl;
      break;
   case 1:
      cout << "Uno!" << endl;
      break;
   case 2:
      cout << "Dos!" << endl;
      break;
   default:
      cout << "Fuera de rango" << endl;
}
```

Ejemplo 2

```
switch (opción)
{
case 0:
case 1:
case 2:
      cout << "Menor de 3";
      break;
case 3:
      cout << "Igual a 3";
      break;
default:
      cout << "Mayor que 3";
}
```

Ejemplo 4.8

Comparación de las sentencias `if-else-if` y `switch`.

Se necesita saber si un determinado carácter `car` es una vocal.

Solución con *if-else-if*

```
if ((car == 'a') || (car == 'A'))
   cout << car << "es una vocal" << endl;
else if ((car == 'e') || (car == 'E'))
   cout << car << "es una vocal" << endl;
else if ((car == 'i') || (car == 'I'))
   cout << car << "es una vocal" << endl;
else if ((car == 'o') || (car == 'O'))
   cout << car << "es una vocal" << endl;
else if ((car == 'u') || (car == 'U'))
   cout << car << "es una vocal" << endl;
else
   cout << car << "no es una vocal" << endl;
```

Solución con *switch*

```
switch (car) {
   case 'a': case 'A':
   case 'e': case 'E':
   case 'i': case 'I':
   case 'o': case 'O':
   case 'u': case 'U':
      cout << car << "es una vocal" << endl;
      break;
   default
      cout << car << "no es una vocal" << endl;
   }
```

Ejemplo 4.9

```
// Programa de ilustración de la sentencia switch
#include <iostream>
using namespace std;

int main()
{
   char nota;
   cout << "Introduzca calificación (A-H) y pulse Intro:";
   cin >> nota;

   switch (nota)
   {
      case 'A':  cout << "Excelente."
                        << "Examen superado\n";
                 break;
      case 'B':  cout << "Notable.";
                 cout << "Suficiencia\n";
                 break;
      case 'C':  cout << "Aprobado\n";
                 break;
```

```
      case 'D':
      case 'F':  cout << "Suspendido\n";
                 break;
      default:
         cout "no es posible esta nota";
   }
   cout << "Final de programa" << endl;
   return 0;
}
```

Cuando se ejecuta la sentencia `switch`, se evalúa `nota`; si el valor de la expresión es igual al valor de una etiqueta, entonces se transfiere el flujo de control a las sentencias asociadas con la etiqueta correspondiente. Si ninguna etiqueta coincide con el valor de `nota` se ejecuta la sentencia `default` y las sentencias que vienen detrás de ella. Normalmente, la última sentencia de las sentencias que vienen después de una `case` es una sentencia `break`. Esta sentencia hace que el flujo de control del programa salte a la última sentencia de `switch`. Si no existiera `break`, se ejecutarían también las sentencias restantes de la sentencia `switch`.

Ejecución de prueba 1

```
Introduzca calificación (A-H) y pulse Intro: A
Excelente. Examen superado
Final de programa
```

Ejecución de prueba 2

```
Introduzca calificación (A-H) y pulse Intro: B
Notable. Suficiencia
Final de programa
```

Ejecución de prueba 3

```
Introduzca calificación (A-H) y pulse Intro: E
No es posible esta nota
Final de programa
```

Precaución

Si se olvida `break` en una sentencia `switch`, el compilador no emitirá un mensaje de error, ya que se habrá escrito una sentencia `switch` correcta sintácticamente, pero no realizará las tareas previstas.

Ejemplo 4.10

```
int tipo_vehículo;
cout << "Introduzca tipo de vehículo:";
cin >> tipo_vehículo, peaje;
```

```
switch(tipo_vehículo)
{
   case 1:
      cout << "turismo";
      peaje = 500;
      break;   ———→  Si se omite esta break, el vehículo primero será turismo y luego
                     autobús.
   case 2:
      cout << "autobús";
      peaje = 3000;
      break;
   case 3:
      cout << "motocicleta";
      peaje = 300;
      break;
   default:
      cout << "vehículo no autorizado";
}
```

Cuando la computadora comienza a ejecutar una sentencia `case,` no detiene su ejecución hasta que se encuentra o bien una sentencia `break` o bien una sentencia `switch`.

4.5.1. Caso particular case

Está permitido tener varias expresiones `case` en una alternativa dada dentro de la sentencia `switch`. Por ejemplo, se puede escribir:

```
switch(c) {
   case '0': case '1': case '2': case '3': case '4':
   case '5': case '6': case '7': case '8': case '9':
      num_digitos++;  // se incrementa en 1 el valor de num_digitos
      break;
   case '': case '\t': case '\n':
      num_blancos++;  // se incrementa en 1 el valor de num_blancos
      break;
   default:
      num_distintos++;
}
```

4.5.2. Uso de sentencias switch en menús

La sentencia `if-else` es más versátil que la sentencia `switch` y se puede utilizar unas sentencias `if-else` anidadas o multidecisión, en cualquier parte que se utiliza una sentencia `case`. Sin embargo, normalmente, la sentencia `switch` es más clara. Por ejemplo, la sentencia `switch` es idónea para implementar menús.

Un *menú* de un restaurante presenta una lista de alternativas para que un cliente elija entre sus diferentes opciones. Un menú en un programa de computadora hace la misma función: presentar una lista de alternativas en la pantalla para que el usuario elija una de ellas.

4.6. EXPRESIONES CONDICIONALES: EL OPERADOR ?:

Las sentencias de selección (`if` y `switch`) consideradas hasta ahora son similares a las sentencias previstas en otros lenguajes, tales como C y Pascal. Sin embargo, C++ ha heredado un tercer mecanismo de selección de su lenguaje raíz C, una expresión que produce uno de dos valores, resultado de una expresión lógica o booleana (también denominada condición). Este mecanismo se denomina *expresión condicional*. Una expresión condicional tiene el formato `C? A : B` y es realmente una operación ternaria (tres operandos) en el que `C`,`A` y `B` son los tres operandos y `?` es el operador.

Sintaxis

condición ? expresión$_1$: expresión$_2$

condición	es una expresión lógica
expresión$_1$/expresión$_2$	son expresiones compatibles de tipos

Se evalúa *condición*, si el valor de *condición* es verdadera (distinto de cero) entonces se devuelve como resultado el valor de *expresión*$_1$; si el valor de *condición* es falsa (cero), se devuelve como resultado el valor de *expresión*$_2$.

Uno de los medios más sencillos del operador condicional (`?:`) es utilizar el operador condicional y llamar a una de dos funciones.

Ejemplos

1.1 `a == b ? función1() : función2();`
es equivalente a la siguiente sentencia:

```
if (a == b)
   función1();
else
   función2();
```

1.2 El operador `?:` se utiliza en el siguiente segmento de código para asignar el menor de dos valores de entrada asignados a `Menor`.

```
int Entrada1;
int Entrada2;
cin >> Entrada1 >> Entrada2;
int Menor = Entrada1 <= Entrada2 ? Entrada1 : Entrada2
```

Ejemplo 4.11

```
#include <iostream>
using namespace std;

void main()
{
   float n1, n2;

   cout << "Introduzca dos números positivos o negativos:";
   cin >> n1 >> n2;
```

```
//if-else
   cout << endl << "if-else";
   if (n1 > n2)
      cout << n1 << " > " << n2;
   else
      cout << n1 << " > " << n2;

   // operador condicional
   cout << endl << condicional:";
   n1 > n2 ? cout << n1 << " > " << n2
           : cout << n1 << " < " << n2;
}
```

4.7. EVALUACIÓN EN CORTOCIRCUITO DE EXPRESIONES LÓGICAS

Cuando se evalúan expresiones lógicas en C++ se puede emplear una técnica denominada *evaluación en cortocircuito*. Este tipo de evaluación significa que se puede detener la evaluación de una expresión lógica tan pronto como su valor pueda ser determinado con absoluta certeza. Por ejemplo, si el valor de `(soltero == 's')` es falso, la expresión lógica `(soltero == 's') && (sexo = 'h') && (edad > 18) && (edad <= 45)` será falsa con independencia de cuál sea el valor de las otras condiciones. La razón es que una expresión lógica del tipo

```
falso && (...)
```

debe ser siempre falsa, cuando uno de los operandos de la operación `AND` es falso. En consecuencia, no hay necesidad de continuar la evaluación de las otras condiciones cuando `(soltero == 's')` se evalúa a falso.

El compilador C++ utiliza este tipo de evaluación. Es decir, la evaluación de una expresión lógica de la forma `a1 && a2` se detiene si la subexpresión `a1` de la izquierda se evalúa a falsa.

C++ realiza evaluación en cortocircuito con los operadores `&&` y `||`, de modo que evalúa primero la expresión más a la izquierda de las dos expresiones unidas por `&&` o bien por `||`. Si de esta evaluación se deduce la información suficiente para determinar el valor final de la expresión (independiente del valor de la segunda expresión), el compilador de C++ no evalúa la segunda expresión.

Ejemplo

Si `x` es negativo, la expresión

```
(x >= 0) && (y > 1)
```

se evalúa en cortocircuito ya que `x >= 0` será falso y, por tanto, el valor final de la expresión será falso.

En el caso del operador `||` se produce una situación similar. Si la primera de las dos expresiones unidas por el operador `||` es *verdadera*, entonces la expresión completa es *verdadera*, con independencia de que el valor de la segunda expresión sea *verdadero* o *falso*. La razón es que el operador `|| OR` produce resultado verdadero si el primer operando es verdadero.

Otros lenguajes, distintos de C++, utilizan evaluación completa. En evaluación completa, cuando dos expresiones se unen por un símbolo `&&` o `||`, se evalúan siempre ambas expresiones y, a continuación, se utilizan las tablas de verdad de `&&` o bien `||` para obtener el valor de la expresión final.

Ejemplo

Si x es cero, la condición

```
if ((x != 0.0) && (y/x > 7.5))
```

es falsa ya que (x != 0.0) es falsa. Por consiguiente, no hay necesidad de evaluar la expresión (y / x > 7.0) cuando x sea cero. Sin embargo, si altera el orden de las expresiones, al evaluar el compilador la sentencia if

```
if ((y / x > 7.5) && (x != 0.0))
```

se produciría un error en tiempo de ejecución de división por cero («division by zero»).

> El orden de las experiencias con operadores && y || puede ser crítico en determinadas situaciones.

4.8. PUESTA A PUNTO DE PROGRAMAS

Estilo y diseño

1. El estilo de escritura de una sentencia if e if-else es el sangrado de las diferentes líneas en el formato siguiente:

```
if (expresión_lógica)
   sentencia1
else
   sentencia2
```

```
if (expresión_lógica)
      {
        sentencia1
           .
        .
        .
     sentenciak
   }
else
   {
      sentenciak+1
         .
         .
         .
      sentencian
   }
```

En el caso de sentencias if-else-if utilizadas para implementar una estructura de selección multialternativa se suele escribir de la siguiente forma:

```
if (expresión_lógica1)
    sentencia1
else if (expresión_lógica2)
    sentencia2
  .
  .
  .
```

```
else if (expresión_lógica_n)
    sentencia_n
else
    sentencia_n+1
```

2. Una construcción de selección múltiple se puede implementar más eficientemente con una estructura `if-else-if` que con una secuencia de sentencias independientes `if`. Por ejemplo,

```
cout <<"Introduzca nota";
cin >> nota
if (nota < 0 || nota > 100)
{
  cout << nota << " no es una nota válida.\n";
  return'?';
}
if (nota >= 90( && (nota <= 100)
  return 'A';
if (nota >= 80) && (nota < 90)
  return 'B';
if (nota >=70) && (nota < 80)
  return 'C';
if (nota >= 60) && (nota < 70)
  return 'D';
if (nota < 60)
  return 'F';
```

Con independencia del valor de `nota` se ejecutan todas las sentencias `if`; 5 de las expresiones lógicas son expresiones compuestas, de modo que se ejecutan 16 operaciones con independencia de la nota introducida. En contraste, las sentencias `if` anidadas reducen considerablemente el número de operaciones a realizar (3 a 7), todas las expresiones son simples y no se evalúan todas ellas siempre.

```
cout << "Introduzca nota";
cin >> nota
if (nota < 0 || nota > 100)
{
   cout << nota << " no es una nota válida.\n";
   return '?';
}
else if (nota >= 90)
   return 'A';
else if (nota >= 80)
   return 'B';
else if (nota >= 70)
   return 'C';
else if (nota >= 60)
   return 'D';
else
   return 'F';
```

4.9. ERRORES FRECUENTES DE PROGRAMACIÓN

1. *Uno de los errores más comunes en una sentencia `if` es utilizar un operador de asignación (`=`) en lugar de un operador de igualdad (`==`).*

2. *En una sentencia* `if` *anidada, cada claúsula* `else` *se corresponde con la* `if` *precedente más cercana. Por ejemplo, en el segmento de programa siguiente.*

```
if (a > 0)
if (10 > 0)
c = a + b;
else
c = a + abs(b);
d = a * b * c;
```

¿Cuál es la sentencia `if` asociada a `else`?

El sistema más fácil para evitar errores es el sangrado o indentación, con lo que ya se aprecia que la cláusula `else` se corresponde a la sentencia que contiene condición `b > 0`.

```
if (a > 0)
   if (b > 0)
      c = a + b;
   else
      c = a + abs(b);
d = a * b * c;
```

3. *Las comparaciones con operadores* `==` *de cantidades algebraicamente iguales pueden producir una expresión lógica falsa, debido a que la mayoría de los números reales no se almacenan exactamente*. Por ejemplo, aunque las expresiones reales siguientes son equivalentes:

```
a * (1 / a)
1.0
```

son algebraicamente iguales, la expresión

```
a * (1 / a) == 1.0
```

puede ser falsa debido a que `a` es real.

4. *Cuando en una sentencia* `switch` *o en un bloque de sentencias falsas una de las llaves (`{`, `}`) aparece un mensaje de error tal como:*

```
Error ...: Cumpound statement missing } in function
```

Si no se tiene cuidado con la presentación de la escritura del código, puede ser muy difícil localizar la llave que falta.

5. *El selector de una sentencia* `switch` *debe ser de tipo entero o compatible entero*. Así, las constantes reales

```
2.4, -4.5, 3.1416
```

no pueden ser utilizadas en el selector.

6. *Cuando se utiliza una sentencia* `switch`*, asegúrese que el selector de* `switch` *y las etiquetas* `case` *son del mismo tipo (*`int`*,* `char` *o* `bool` *pero no* `float`*). Si el selector se evalúa a un valor no listado en ninguna de las etiquetas* `case`*, la sentencia* `switch` *no gestionará ninguna acción; por esta causa se suele poner una etiqueta* `default` *para resolver este problema.*

RESUMEN

Sentencia `if`

Una alternativa

```
if (a != 0)
   resultado = a/b;
```

Múltiples alternativas

```
if (x < 0)
{
   cout << "Negativo" << endl;
   abs_x = -x;
}
else if (x == 0)
{
   cout << "Cero" << endl;
   abs_x = 0;
}
else
{
   cout << "Positivo" << endl;
   abs_x = x;
}
```

Dos alternativas

```
if (a >= 0)
   cout << a << " es positivo" << endl;
else
   cout << a << " es negativo" << endl;
```

Sentencia `switch`

```
switch (sig_car)
{
   case 'A': case'a':
      cout << "Sobresaliente" << endl;
      break;
   case 'B': case 'b':
      cout << "Notable" << endl;
      break;
   case 'C': case 'c':
      cout << "Aprobado" << endl;
      break;
   case 'D': case 'd':
      cout << "Suspenso" << endl;
      break;
   default
      cout << "nota no válida" << endl;
   }  // fin de switch
```

EJERCICIOS

4.1. ¿Qué valor se asigna a consumo en la sentencia `if` siguiente si velocidad es 120?

```
if (velocidad > 80)
     consumo = 10.00;
else if (velocidad > 100)
     consumo = 12.00;
else if (velocidad > 120)
     consumo = 15.00;
```

4.2. Explique las diferencias entre las sentencias de la columna de la izquierda y de la columna de la derecha. Para cada una de ellas deducir el valor final de x si el valor inicial de x es 0.

```
if (x >= 0)          if (x >= 0)
     x = x+1;             x = x+1;
else if (x >= 1);    if (x >= 1)
     x = x+2;             x = x+2;
```

4.3. ¿Qué salida producirá el siguiente código cuando se inserta en un programa completo?

```
int x = 2;
cout << "Arranque\n";
if (x <= 3)
if (x != 0)
    cout << "Hola desde el segundo if.\n";
else
    cout << "Hola desde el else.\n";
cout << "Fin\n";
cout << "Arranque de nuevo\n";
if (x > 3)
       if (x != 0)
          cout << "Hola desde el segundo
            if.\n";
       else
          cout << "Hola desde el else.\n";
    cout << "De nuevo fin\n";
```

4.4. ¿Qué hay de incorrecto en el siguiente código?

```
if (x = 0) cout << x << " = 0\n";
else cout << x << " != 0\n";
```

4.5. ¿Cuál es el error del siguiente código?

```
if (x < y < z) cout << x << "<" << y
   << "<" << z << endl;
```

4.6. ¿Cuál es el error de este código?

```
cout << "Introduzca n:";
cin >> n;
if (n < 0)
      cout << "Este número es nega-
         tivo. Pruebe de nuevo .\n";
      cin  >> n;
else
      cout << "conforme. n= " << n
<< endl;
```

4.7. Escribir una sentencia if-then-else que clasifique un entero *x* en una de las siguientes categorías y escriba un mensaje adecuado:

x < o *o bien* 0 ℑ x ℑ 100 *o bien* x > 100

4.8. Se trata de escribir un programa que clasifique enteros leídos del teclado de acuerdo a los siguientes puntos: 1. Si es 30 o mayor, o negativo, visualizar un mensaje en ese sentido; en caso contrario, si es un nuevo primo, potencia de 2, o un número compuesto, visualizar el mensaje correspondiente; si son cero o 1, visualizar 'cero' o 'unidad'.

4.9. Escribir un programa que determine el mayor de tres números.

4.10. El domingo de Pascua es el primer domingo después de la primera luna llena posterior al equinoccio de primavera, y se determina mediante el siguiente cálculo sencillo:

```
A = año mod 19
B = año mod 4
C = año mod 7
D = (19 * A + 24) mod 30
E = (2 * B + 4 * C + 6 * D + 5) mod
7
N = (22 + D + E)
```

Donde N indica el número de día del mes de marzo (si N es igual o menor que 3) o abril (si es mayor que 31). Construir un programa que determine fechas de domingos de Pascua.

4.11. Codificar un programa que escriba la calificación correspondiente a una nota, de acuerdo con el siguiente criterio:

0 ⩽ a < 5.0	Suspenso
5 ⩽ a ℑ 6.5	Aprobado
6.5 [a ℑ 8.5	Notable
8.5 [a < 10	Sobresaliente
10 =	Matrícula de honor

4.12. Determinar si el carácter asociado a un código introducido por teclado corresponde a un carácter alfabético, dígito, de puntuación, especial o no imprimible.

PROBLEMAS

4.1. Un archivo contiene dos fechas en el formato día (1 a 31), mes (1 a 12) y año (entero de cuatro dígitos), correspondientes a la fecha de nacimiento y la fecha actual, respectivamente. Escribir un programa que calcule y visualice la edad del individuo. Si es la fecha de un bebé (menos de un año de edad), la edad se debe dar en meses y días; en caso contrario, la edad se calculará en años.

4.2. Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo 1984). Sin embargo, los años múltiplos de 100 sólo son bisiestos cuando a la vez son múltiples de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo será).

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 67).

2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

4.1. *¿Qué errores de sintaxis tiene la siguiente sentencia?*

```
if  x > 25.0
    y = x
else
    y = z;
```

4.2. *¿Qué valor se asigna a* `consumo` *en la sentencia* `if` *siguiente si velocidad es* `120`*?*

```
if (velocidad > 80)
 consumo = 10.00;
else if (velocidad > 100)
 consumo = 12.00;
else if (velocidad > 120)
 consumo = 15.00;
```

4.3. *¿Qué salida producirá el código siguiente, cuando se inserta en un programa completo?*

```
int primera_opcion = 1;
switch (primera_opcion + 1)
{
 case 1:
      cout << "Cordero asado\n";
      break;
 case 2:
      cout << "Chuleta lechal\n";
      break;
 case 3:
      cout << "Chuletón\n";
 case 4:
      cout << "Postre de pastel\n";
      break;
 default:
      cout << "Buen apetito\n";
}
```

4.4. *¿Qué salida producirá el siguiente código, cuando se inserta en un programa completo?*

```
int x = 2;

cout << "Arranque\n";
if (x <= 3)
        if (x != 0)
             cout << "Hola desde el
                segundo if.\n";
        else
             cout << "Hola desde el
                else.\n";
cout << "Fin\n";
cout << "Arranque de nuevo\n";
if (x > 3)
        if (x != 0)
             cout << "Hola desde el
                segundo if.\n";
        else
             cout << "Hola desde el
                else.\n";
cout << "De nuevo fin\n";
```

4.5. *Escribir una sentencia* `if-else` *que visualice la palabra* `Alta` *si el valor de la variable nota es mayor que* `100` *y* `Baja` *si el valor de esa nota es menor que* `100`.

4.6. *¿Cuál es la salida de este segmento de programa?*

```
int x = 1;

cout << x << endl;
{
     cout << x << endl;
     int x = 2;
     cout << x << endl;
     {
          cout << x << endl;
          int x = 3;
          cout << x << endl;
     }
     cout << x << endl;
}
```

4.7. *Escribir una sentencia* `if-else` *que clasifique un entero x en una de las siguientes categorías y escriba un mensaje adecuado:*

`x < 0`   o bien   `0 ℑ x ℑ 100`   o bien   `x > 100`

4.8. *Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo 1984). Sin embargo, los años múltiplos de 100 sólo son bisiestos cuando a la vez son múltiplos de 400 (por ejemplo, 1800 no es bisiesto, mientras que 2000 sí lo es).*

PROPLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 69).

2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

4.1. *Escribir un programa que introduzca el número de un mes (1 a 12) y el año y visualice el número de días de ese mes.*

4.2. *Cuatro enteros entre 0 y 100 representan las puntuaciones de un estudiante de un curso de informática. Escribir un programa para encontrar la media de estas puntuaciones y visualizar una tabla de notas de acuerdo al siguiente cuadro:*

Media	Puntuación
[90-100]	A
[80-90)	B
[70-80)	C
[60-70)	D
[0-60)	E

4.3. *Se desea calcular el salario neto semanal de los trabajadores de una empresa de acuerdo a las siguientes normas:*

Horas semanales trabajadas < = 38, a una tasa dada.
Horas extras (38 o más), a una tasa 50 por 100 superior a la ordinaria.
Impuestos 0 por 100, si el salario bruto es menor o igual a 300 euros.
Impuestos 10 por 100, si el salario bruto es mayor de 300 euros.

4.4. *Escribir un programa que lea dos números enteros y visualice el menor de los dos.*

4.5. *Escribir y comprobar un programa que resuelva la ecuación cuadrática* ($ax^2 + bx + c = 0$).

4.6. *Escribir un programa que lea tres enteros y emita un mensaje que indique si están o no en orden numérico.*

4.7. *Escribir un programa que lea los valores de tres lados posibles de un triángulo a, b y c, y calcule en el caso de que formen un triángulo su área y su perímetro, sabiendo que su área viene dada por la siguiente expresión:*

$$\text{Área} = \sqrt{p\,(p - a)(p - b)(p - c)}$$

donde p es el semiperímetro del triángulo $p = (a + b + c)/2$

4.8. *Escribir y ejecutar un programa que simule un calculador simple. Lee dos enteros y un carácter. Si el carácter es un +, se visualiza la suma; si es un –, se visualiza la diferencia; si es un *, se visualiza el producto; si es un /, se visualiza el cociente; y si es un % se imprime el resto.*

4.9. *Escribir un programa que calcule los ángulos agudos de un triángulo rectángulo a partir de las longitudes de los catetos.*

CAPÍTULO 5

Estructuras de control: bucles

Contenido

INTRODUCCIÓN

Una de las características de las computadoras que aumentan considerablemente su potencia es su capacidad para ejecutar una tarea muchas (*repetidas*) veces con gran velocidad, precisión y fiabilidad. Las tareas repetitivas es algo que los humanos encontramos difíciles y tediosas de realizar. En este capítulo se estudian las *estructuras de control iterativas* o *repetitivas* que realizan la repetición o iteración de acciones. C++ soporta tres tipos de estructuras de control: los bucles **`while`**, **`for`** y **`do-while`**. Estas estructuras de control o sentencias repetitivas controlan el número de veces que una sentencia o listas de sentencias se ejecutan.

CONCEPTOS CLAVE

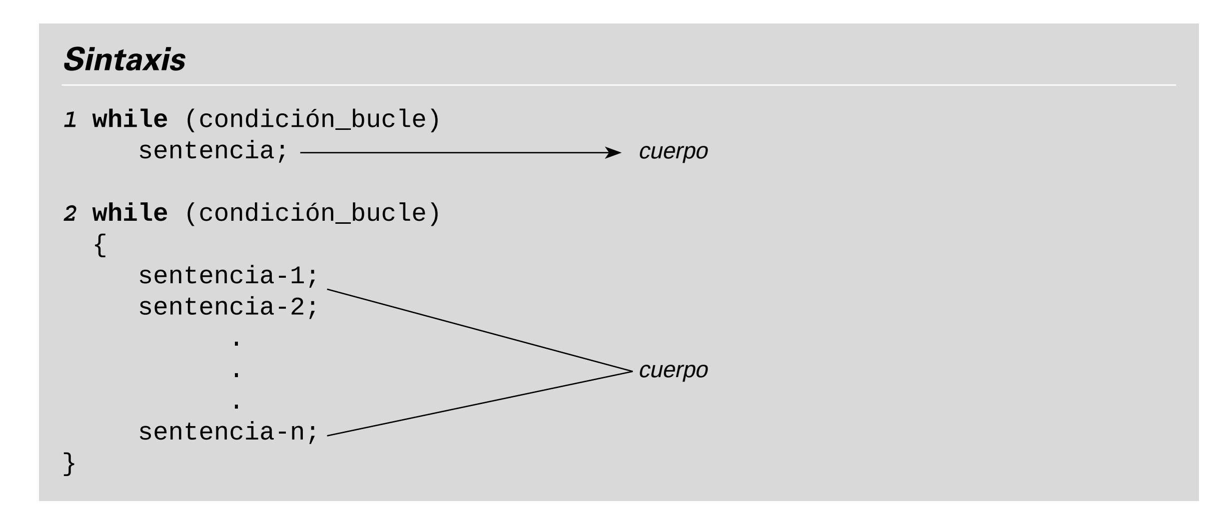
- Bucle.
- Comparación de `while`, `for` y `do`.
- Control de bucles.
- Iteracción/repetición.
- Optimización de bucles.
- Sentencia **`break`**.
- Sentencia **`do-while`**.
- Sentencia **`for`**.
- Sentencia **`while`**.
- Terminación de un bucle.

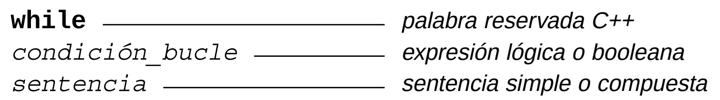
5.1. LA SENTENCIA `WHILE`

Un bucle es cualquier construcción de programa que repite una sentencia o secuencia de sentencias un número de veces. La sentencia (o grupo de sentencias) que se repiten en un bloque se denomina **cuerpo** del bucle y cada repetición del cuerpo del bucle se llama **iteración** del bucle. Las dos principales cuestiones de diseño en la construcción del bucle son: ¿Cuál es el cuerpo del bucle? ¿Cuántas veces se iterará el cuerpo del bucle?

Un bucle `while` tiene una *condición* del bucle (una expresión lógica) que controla la secuencia de repetición. La posición de esta condición del bucle es delante del cuerpo del bucle y significa que un bucle `while` es un bucle *pretest* de modo que cuando se ejecuta el mismo, se evalúa la condición *antes* de que se ejecute el cuerpo del bucle. La Figura 5.1 representa el diagrama del bucle `while`.

El diagrama de la Figura 5.1 indica que la ejecución de la sentencia o sentencias expresadas se repite *mientras* la condición del bucle permanece verdadera y termina cuando se hace falsa. También indica el diagrama anterior que la condición del bucle se evalúa antes de que se ejecute el cuerpo del bucle y, por consiguiente, si esta condición es inicialmente falsa, el cuerpo del bucle no se ejecutará. En otras palabras, el cuerpo de un bucle `while` se ejecutará *cero* o *más veces*.





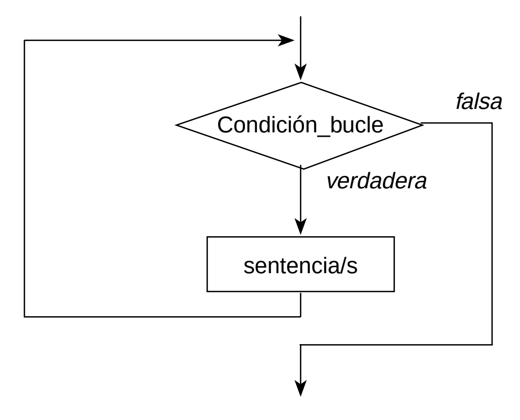


Figura 5.1. Diagrama de flujo de un bucle `while`.

El *comportamiento* o *funcionamiento* de una sentencia (bucle) `while` es:

1. Se evalúa la *`condición_bucle`*.
2. Si *`condición_bucle`* es verdadera:
   a) La *`sentencia`* especificada, denominada **cuerpo** del bucle, se ejecuta.
   b) Vuelve el control al paso 1.
3. En caso contrario:
   El control se transfiere a la sentencia siguiente al bucle o sentencia `while`.

El cuerpo del bucle se repite **mientras** que la expresión lógica (condición del bucle) sea verdadera. Cuando se evalúa la expresión lógica y resulta falsa, se termina y se *sale* del bucle y se ejecuta la siguiente sentencia del programa después del cuerpo de la sentencia `while`.

```
// cuenta hasta 10 (1 a 10)
int x = 1;
while (x <= 10)
   cout << "x: " << ++x;
```

Ejemplo

```
// visualizar n asteriscos
contador = 0;               <-- inicialización
while (contador < n)        <-- prueba/condición
{
   cout << " * ";
   contador++;              <-- actualización (incrementa en
                                                1 contador)
} // fin de while
```

La variable que representa la condición del bucle se denomina también **variable de control del bucle** debido a que su valor determina si el cuerpo del bucle se repite. La variable de control del bucle debe ser: (1) inicializada, (2) comprobada, y (3) actualizada para que el cuerpo del bucle se ejecute adecuadamente. Cada etapa se resume así:

1. *Inicialización*. `contador` se establece a un valor inicial (se inicializa a cero, aunque podría ser otro su valor) antes de que se alcance la sentencia `while`.
2. *Prueba/condición*. Se comprueba el valor de `contador` antes de que comience la repetición de cada bucle (denominada *iteración* o *pasada*).
3. *Actualización*. `Contador` se actualiza (su valor se incrementa en 1, mediante el operador ++) durante cada iteración.

Si la variable de control no actualiza el bucle, éste se ejecutará «siempre». Tal bucle se denomina **bucle infinito**. En otras palabras, un bucle infinito (sin terminación) se producirá cuando la condición del bucle permanece y no se hace falsa en ninguna iteración.

```
// bucle infinito
contador = 1;
while (contador < 100)
{
   cout << contador << endl;
   contador--;              <-- decrementa en 1 contador
}
```

`contador` se inicializa a 1 (menor de 100) y como `contador--` decrementa en 1 el valor de `contador` en cada iteración, el valor del contador nunca llegará a valer 100, valor necesario de `contador` para que la condición del bucle sea falsa. Por consiguiente, la condición `contador < 100` siempre será verdadera, resultando un bucle infinito, cuya salida será:

```
 1
 0
-1
-2
-3
-4
 .
 .
 .
```

Ejemplo

```
// Bucle de muestra con while

int main()
{
   int contador = 0;    // inicializa la condición

   while(contador < 5)  // condición de prueba
   {
      contador ++;        // cuerpo del bucle
      cout << "contador: " << contador << "\n";
   }

   cout << "Terminado.Contador: " << contador << "\n";
   return 0;
}
```

Ejecución

```
contador:  1
contador:  2
contador:  3
contador:  4
contador:  5
Terminado.Contador: 5
```

5.1.1. Operadores de incremento y decremento (++,--)

C++ ofrece los operadores de incremento (++) y decremento (--) que soporta una sintaxis abreviada para añadir (incrementar) o restar (decrementar) `1` al valor de una variable. Recordemos del Capítulo 3 la sintaxis de ambos operadores:

```
++nombreVariable            // preincremento
nombreVariable++            // postincremento
```

```
--nombreVariable            // predecremento
nombreVariable--            // postdecremento
```

Ejemplo 5.1

Si `i` *es una variable entera cuyo valor es 3, las variables* `k` *e* `i` *toman los valores sucesivos que se indican en las sentencias siguientes:*

```
k = i++;            // asigna el valor 3 a k y 4 a i
k = ++i;            // asigna el valor 5 a k y 5 a i
k = i--;            // asigna el valor 5 a k y 4 a i
k = --i;            // asigna el valor 4 a k y 3 a i
```

Ejemplo 5.2

Uso del operador de incremento ++ para controlar la iteración de un bucle (una de las aplicaciones más usuales de ++).

```
// programa cálculo de calorías
#include <iostream>
using namespace std;

int main()
{
   int num_de_elementos, cuenta,
       calorias_por_alimento, calorias_total;

   cout << "¿Cuántos alimentos ha comido hoy?";
   cin >> num_de_elementos;

   calorias_total = 0;
   cuenta = 1;
   cout << "Introducir el número de calorias de cada uno de los"
        << num_elementos << " alimentos tomados:\n";

   while (cuenta++ <= numero_de_elementos)
   {
      cin >> calorias_por_alimento;
      calorias_total = calorias_total + calorias_por_alimento;
   }

   cout << "Las calorias totales consumidas hoy son = "
        << calorias_total << endl;
   return 0;
}
```

Ejecución de muestra

```
¿Cuántos alimentos ha comido hoy? 8.
Introducir el número de calorías de cada uno de los 8 alimentos tomados:
500   50   1400   700   10   5   250   100
Las calorías totales consumidas hoy son = 3015
```

5.1.2. Terminaciones anormales de un ciclo

Un error típico en el diseño de una sentencia `while` se produce cuando el bucle sólo tiene una sentencia en lugar de varias sentencias como se planeó. El código siguiente:

```
contador = 1;
while (contador < 25)
   cout << contador << endl;
   contador++;
```

visualizará infinitas veces el valor 1. Es decir, entra en un bucle infinito del que nunca sale porque no se actualiza (modifica) la variable de control `contador`.

La razón es que el punto y coma al final de la línea `cout << contador << endl;` hace que el bucle termine en ese punto y coma, aunque aparentemente el sangrado pueda dar la sensación de que el cuerpo de `while` contiene dos sentencias: `cout` ... y `contador++;`.

El error se hubiera detectado rápidamente si el bucle se hubiera escrito correctamente con una sangría.

```
contador = 1;
while (contador < 25)
   cout << contador << endl;
contador++;
```

La solución es muy sencilla, utilizar las llaves de la sentencia compuesta:

```
contador = 1;
while (contador < 25)
{
   cout << contador << endl;
   contador++;
}
```

5.1.3. Diseño eficiente de bucles

Una cosa es analizar la operación de un bucle y otra diseñar eficientemente sus propios bucles. Los principios a considerar son: primero, analizar los requisitos de un nuevo bucle con el objetivo de determinar su inicialización, prueba (condición) y actualización de la variable de control del bucle; segundo, desarrollar *patrones estructurales* de los bucles que se utilizan frecuentemente.

5.1.4. Bucles `while` con cero iteraciones

El cuerpo de un bucle no se ejecuta nunca si la prueba o condición de repetición del bucle no se cumple (es falsa) cuando se alcanza `while` la primera vez.

```
contador = 10
while (contador > 100)
{
   ...
}
```

El bucle anterior nunca se ejecutará ya que la condición del bucle (`contador > 100`) es falsa la primera vez que se ejecuta. El cuerpo del bucle nunca se ejecutará.

5.1.5. Bucles controlados por centinela

Normalmente, no se conoce con exactitud cuántos elementos de datos se procesarán antes de comenzar su ejecucción. Esto se produce bien porque hay muchos datos a contar manualmente o porque el número de datos a procesar depende de cómo prosigue el proceso de cálculo.

Un medio para manejar esta situación es instruir al usuario a introducir un único dato definido y especificado denominado *valor centinela* como último dato. La condición del bucle comprueba cada dato y termina cuando se lee el valor centinela. El valor centinela se debe seleccionar con mucho cuidado y debe ser un valor que no pueda producirse como dato. En realidad el centinela es un valor que sirve para terminar el proceso del bucle.

```
// entrada de datos numéricos
// centinela –1
const int centinela = -1
cout << "Introduzca primera nota";
cin >> nota;
while (nota != centinela)
{
   cuenta++;
   suma += nota;
   cout << "Introduzca la siguiente nota:";
   cin >> nota;
} // fin de while
cout << "Final"
```

Si se lee el primer valor de `nota`, por ejemplo 25, y luego se ejecuta el bucle, la salida podría ser ésta:

```
Introduzca primera nota: 25
Introduzca siguiente nota: 30
Introduzca siguiente nota: 90
Introduzca siguiente nota: -1      // valor del centinela
Final
```

5.1.6. Bucles controlados por indicadores (banderas)

Las variables tipo `bool` se utilizan con frecuencia como *indicadores* o *banderas de estado* para controlar la ejecución de un bucle. El valor del indicador se inicializa (normalmente a falso `"false"`)antes de la entrada al bucle y se redefine (normalmente a verdadero `"true"`) cuando un suceso específico ocurre dentro del bucle. Un *bucle controlado por bandera* o *indicador* se ejecuta hasta que se produce el suceso anticipado y se cambia el valor del indicador.

Ejemplo 5.3

Se desea leer diversos datos tipo carácter introducidos por teclado mediante un bucle `while` y se debe terminar el bucle cuando se lea un dato tipo dígito (rango '0'a '9').

La variable bandera `digito_leido` se utiliza como un indicador que representa cuando un dígito se ha introducido por teclado.

Variable bandera	*Significado*
`digito_leido`	Su valor es falso antes de entrar en el bucle y mientras el dato leído sea un carácter y es verdadero cuando el dato leído es un dígito.

El problema que se desea resolver es la lectura de datos carácter y la lectura debe detenerse cuando el dato leído sea numérico (un dígito de '0' a '9'). Por consiguiente, antes de que el bucle se ejecute y se lean los datos de entrada, la variable `digito_leido` se inicializa a falso (`false`). Cuando se ejecuta el bucle, éste debe continuar ejecutándose mientras el dato leído sea un carácter y, en consecuencia, la variable `digito_leido` toma el valor falso y se debe detener el bucle cuando el dato leído sea un dígito y, en este caso, el valor de la variable `digito_leido` se debe cambiar a verdadero. En consecuencia, la condición del bucle debe ser `¡digito_leido` ya que esta condición es verdadera cuando `digito_leido` es falso. El bucle `while` será similar a:

```
char car;
digito_leido = false;       // no se ha leido ningún dato
while (!digito_leido)
{
   cout << "Introduzca un caracter:";
   cin >> car;
   digito_leido = (('0'<= car) && (car <= '9'));
   ...
}  // fin de while
```

El bucle funciona de la siguiente forma:

1. Entrada del bucle: la variable `digito_leido` tiene por valor «falso».
2. La condición del bucle `!digito_leido` es verdadera, por consiguiente se ejecutan las sentencias del interior del bucle.
3. Se introduce por teclado un dato que se almacena en la variable `car`. Si el dato leído es un carácter la variable `digito_leido` se mantiene con valor falso, ya que ése es el resultado de la sentencia de asignación.

   ```
   digito_leido = (('0'<= car) && (car <= '9'));
   ```

   Si el dato leído es un dígito, entonces la variable `digito_leido` toma el valor verdadero, resultante de la sentencia de asignación anterior.
4. El bucle se termina cuando se lee un dato tipo dígito (`'0'` a `'9'`) ya que la condición del bucle es falsa.

Modelo de bucle controlado por un indicador

El formato general de un bucle controlado por indicador es el siguiente:

1. Establecer el indicador de control del bucle a «falso» o «verdadero» con el objeto de que se ejecute el bucle `while` correctamente la primera vez (normalmente se suele inicializar a «falso»).
2. Mientras la condición de control del bucle sea verdadera:
   2.1. Realizar las sentencias del cuerpo del bucle.
   2.2. Cuando se produzca la condición de salida (en el ejemplo anterior, que el dato carácter leído fuese un dígito) se cambia el valor de la variable indicador o bandera, con el objeto de que entonces la condición de control se haga falsa y, por tanto, el bucle se termina.
3. Ejecución de las sentencias siguientes al bucle.

5.1.7. Bucles `while` (true)

La condición que se comprueba en el bucle `while` puede ser cualquier expresión válida C++. Mientras que la condición permanezca *verdadera* (cierta), el bucle `while` continuará ejecutándose. Se puede crear un bucle que nunca termine utilizando el valor `true` (verdadero) para la condición que se comprueba.

```
1:    //Listado while (true)
2:    #include <iostream>
3:    using namespace std;
4:    int main()
5:    {
6:       int contador = 0;

7:       while (contador = 0);  //también, while (true)
8:       {
9:          contador++;
10:          if (contador > 10)
11:            break;
12:      }
13:      cout << "Contador: " << contador << "\n";
14:      return 0;
15:    }
```

Salida

```
Contador: 11
```

Análisis

En la línea 6, un bucle `while` se establece con una condición que nunca puede ser falsa. El bucle incrementa la variable `contador` en la línea 8 y, a continuación, la línea 9 comprueba a ver si el contador es mayor que 10. Si no es así, el bucle se itera de nuevo. Si `contador` es mayor que 10, la sentencia `break` de la línea 10 termina el bucle `while`, y la ejecución del programa pasa a la línea 12.

Ejercicio 5.1

Calcular la media aritmética de seis números.

El cálculo típico de una media de valores numéricos es: leer sucesivamente los valores, sumarlos y dividir la suma total por el número de valores leídos. El código más simple podría ser:

```
float Num1;
float Num2;
float Num3;
float Num4;
float Num5;
float Num6;
cin >> Num1 >> Num2 >> Num3 >> Num4 >> Num5 >> Num6;
float Media = (Num1+Num2+Num3+Num4+Num5+Num6)/6;
```

Evidentemente, si en lugar de 6 valores fueran 1.000, la modificación del código no sólo sería enorme de longitud sino que la labor repetitiva de escritura sería tediosa. Por ello, la necesidad de utilizar un bucle. El algoritmo más simple sería:

```
definir número de elementos como constante de valor 6
Inicializar contador de números
Inicializar acumulador (sumador) de números
Mensaje de petición de datos
mientras no estén leídos todos los datos hacer
   Leer número
   Acumular valor del número a variable acumulador
   Incrementar contador de números
fin_mientras
Calcular media (Acumulador/Total número)
Visualizar valor de la media
Fin
```

El código en C++ es:

```
// Cálculo de la media de seis números
#include <iostream>
#include <string>
using namespace std;

int main()
{
   const int TotalNum = 6;
   int ContadorNum = 0;
   float SumaNum = 0;
   cout << " Introduzca " << TotalNum << " números " << endl;
   while (ContadorNum < TotalNum)
   {
      // valores a procesar
      float Numero;
      cin >> Número;     // leer siguiente número
      SumaNum += Número; // añadir valor a Acumulador
      ++ContadorNum;     // incrementar números leídos
}
   float Media = SumaNum / ContadorNum;
   cout << "Media:" << Media << endl;
   return 0;
}
```

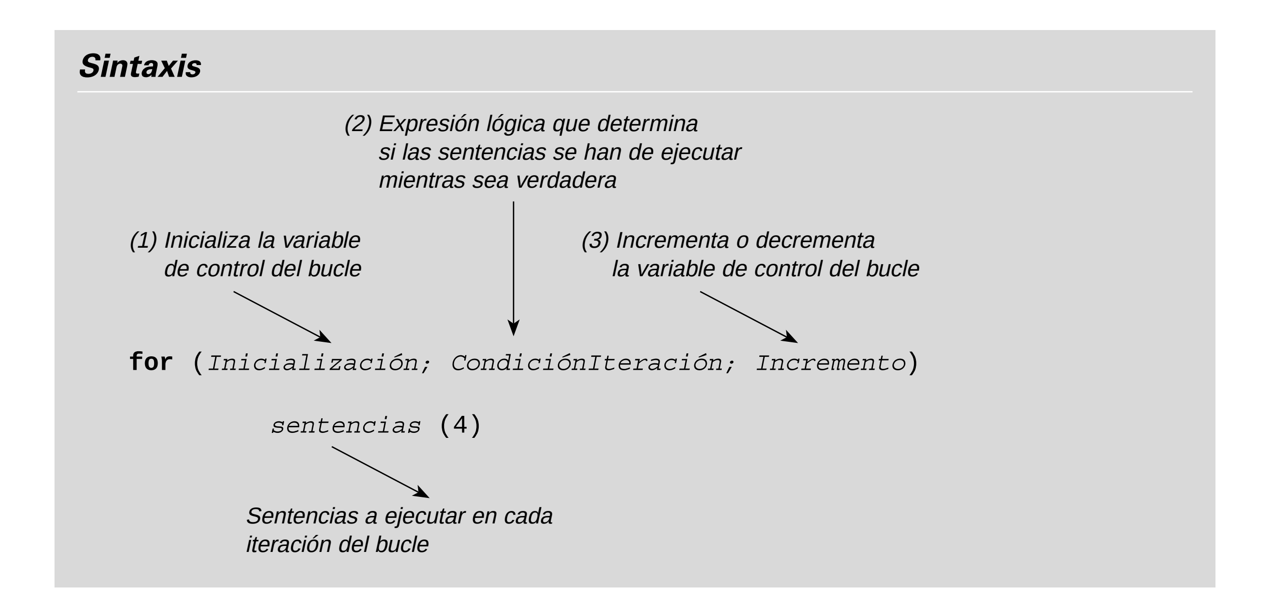
5.2. REPETICIÓN: EL BUCLE FOR

C++ ha heredado la notación de iteración/repetición de C y la ha mejorado ligeramente. El bucle `for` de C++ es superior a los bucles `for` de otros lenguajes de programación tales como BASIC, Pascal y C, ya que ofrece más control sobre la inicialización e incrementación de las variables de control del bucle.

Además del bucle `for`, C++ proporciona otros dos tipos de bucles, `for` y `do`. El bucle `for` que se estudia en esta sección es el más adecuado para implementar *bucles controlados por contador,* que son bucles en los que un conjunto de sentencias se ejecutan una vez por cada valor de un rango especificado, de acuerdo al algoritmo:

```
por cada valor de una variable_contador de un rango específico:
    ejecutar sentencias
```

La sentencia **`for`** (bucle **`for`**) es un método para ejecutar un bloque de sentencias un número fijo de veces. El bucle `for` se diferencia del bucle `while` en que las operaciones de control del bucle se sitúan en un solo sitio: la cabecera de la sentencia.



El bucle `for` contiene las cuatro partes siguientes:

- *Parte de inicialización*, que inicializa las variables de control del bucle. Se pueden utilizar variables de control del bucle simples o múltiples.
- *Parte de iteración*, que contiene una expresión lógica que hace que el bucle realice las iteraciones de las sentencias, mientras que la expresión sea verdadera.
- *Parte de incremento*, que incrementa o decrementa la variable o variables de control del bucle.
- *Sentencias*, acciones o sentencias que se ejecutarán por cada iteración del bucle.

La sentencia `for` es equivalente al siguiente código `while`:

```
inicialización;
while (condiciónIteración)
{
    sentencias del bucle for
    incremento;
}
```

Ejemplo

```
// imprimir Hola 10 veces
for (int i = 0; i < 10; i++)
   cout << "Hola!";
```

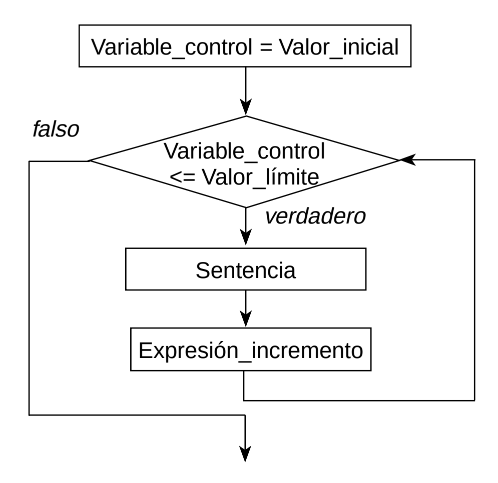


Figura 5.2. Diagrama de sintaxis de un bucle `for`.

Ejemplo

```
for (int i = 0; i < 10; i++)
{
   cout << "Hola!" << endl;
   cout << "el valor de i es: " << i << endl;
}
```

El diagrama de sintaxis de la sentencia `for` es el que se muestra en la Figura 5.2.

Existen dos formas de implementar la sentencia `for` que se utilizan normalmente para implementar bucles de conteo: *formato ascendente*, en el que la variable de control se incrementa y *formato descendente*, en el que la variable de control se decrementa.

```
for (int var_control = valor_inicial; var_control <=
     valor_límite; exp_incremento)
   sentencia
```

formato ascendente *formato descendente*

```
for (int varcontrol = valor_inicial; var_control <=
     valor_límite; exp_decremento)
   sentencia
```

Ejemplo de formato ascendente

```
for (int n = 1; n <= 10; n++)
   cout << n << '\t' << n * n << endl;
```

La variable de control es `n` y su valor inicial es `1` de tipo entero, el valor límite es `10` y la expresión de incremento es `n++`. Esto significa que el bucle ejecuta la sentencia del cuerpo del bucle una vez por

cada valor de `n` en orden ascendente `1` a `10`. En la primera iteración (pasada) `n` tomará el valor 1; en la segunda iteración el valor `2`, y así sucesivamente hasta que `n` toma el valor 10. La salida que se producirá al ejecutarse el bucle será:

```
1    1
2    4
3    9
4    16
5    25
6    36
7    49
8    64
9    81
10   100
```

Ejemplo de formato descendente

```
for (int n = 10; n > 5; n--)
   cout << n << '\t'<< n * n << endl;
```

La salida de este bucle es:

```
10  100
9   81
8   64
7   49
6   36
```

debido a que el valor inicial de la variable de control es `10`, y el límite que se ha puesto es `n > 5` (es decir, verdadera cuando `n = 10, 9, 8, 7, 6`); la expresión de decremento es el operador de decremento `n--`, que decrementa en 1 el valor de `n` tras la ejecución de cada iteración.

Otros intervalos de incremento/decremento

Los rangos de incremento/decremento de la variable o expresión de control del bucle pueden ser cualquier valor y no siempre `1`, es decir, `5, 10, 20, -4`, ..., dependiendo de los intervalos que se necesiten. Así, el bucle

```
for (int n = 0; n <= 100; n += 20)
   cout << n << '\t' << n * n << endl;
```

utiliza la expresión de incremento

```
n += 20
```

que incrementa el valor de `n` en 20, dado que equivale a `n = n + 20`. Así, la salida que producirá la ejecución del bucle es:

```
0      0
20     400
40     1600
60     3600
80     6400
100    10000
```

Ejemplos

```
// ejemplo 1
for (i = 0; i < 10; i++)
   cout << i << "\n";

// ejemplo 2
for (i = 9; i >= 0; i -= 3)
   cout << ( i * i) << "\n";

// ejemplo 3
for (int i = 1; i < 100; i++)
   cout << i << "\n";

// ejemplo 4
for (int i = 0, j = MAX, i < j; i++, j--)
   cout << (i + 2 * j) << endl;
```

El primer ejemplo inicializa la variable de control del bucle `i` a `0` e itera mientras que el valor de la variable `i` es menor que 10. La parte de incremento del bucle incrementa el valor de la variable en `1`. Por consiguiente, el bucle se realiza diez veces con valores de la variable `i` que van de `0` a `9`.

El segundo ejemplo muestra un bucle descendente que inicializa la variable de control a `9`. El bucle se realiza mientras que `i` no sea negativo, como la variable se decrementa en `3`, el bucle se ejecuta cuatro veces con el valor de la variable de control `i, 9, 6, 3` y `0`.

El Ejemplo 3 muestra la característica de C++, ya explicada, de que se puede declarar e inicializar la variable de control `i`, en este caso a `1`. La variable se incrementa en 1, por consiguiente, el bucle realiza 99 iteraciones, para `i` de 1 a 99.

El Ejemplo 4 declara dos variables de control, `i` y `j,` y las inicializa a `0` y la constante `MAX`. El bucle se ejecutará mientras `i` sea menor que `j`. Las variables de control `i, j,` se incrementan ambas en `1`.

Ejemplo 5.4

Suma de los 10 primeros números.

```
// demo de un bucle for
#include <iostream>
using namespace std;

int main()
{
   int suma = 0;
   for (int n = 1; n <= 10; n++)
      suma = suma + n;

   cout << "La suma de los números 1 a 10 es "
        << suma << endl;
   return 0;
}
```

La salida del programa es

```
La suma de los números 1 a 10 es 55
```

5.2.1. Diferentes usos de bucles `for`

ANSI C++ Estándar requiere que las variables sean declaradas en la expresión de inicialización de un bucle `for` sean locales al bloque del bucle `for`. De igual modo permite que:

- El valor de la variable de control se puede modificar en valores diferentes de 1.
- Se pueden utilizar más de una variable de control.

Ejemplo de declaración local de variables de control

Cuando un bucle `for` declara una variable de control, esa variable permanece hasta el final del ámbito que contiene a `for`. Por consiguiente, el siguiente código contiene sentencias válidas.

```
for (int i != 0; i < 10; i++)
   cout << i << endl;
cout << " valor de la variable de control"
     << " después de que termina el bucle es " << i << endl;
```

Ejemplos de incrementos/decrementos con variables de control diferentes

Las variables de control se pueden incrementar o decrementar en valores de tipo `int`, pero también es posible en valores de tipo `double` y, en consecuencia, se incrementaría o decrementaría en una cantidad decimal.

```
int n;
for ( n = 1; n <= 10; n = n + 2)
   cout << "n es ahora igual a " << n << endl;

for (n = 0; n >= 100; n = n - 5)
   cout << " n es ahora igual a " << n << endl;

for (double long = 0.75; long <= 5; long = long + 0.05)
   cout << " longitud es ahora igual a " << long << endl;
```

La expresión de incremento en ANSI C++ no necesita ser una suma o una resta. Tampoco se requiere que la inicialización de una variable de control sea igual a una constante. Se puede inicializar y cambiar una variable de control del bucle en cualquier cantidad que se desee. Naturalmente, cuando la variable de control no sea de tipo `int`, se tendrán menos garantías de precisión. Por ejemplo, el siguiente código muestra un medio más para arrancar un bucle `for`.

```
for (double x = pow(y, 3.0); x > 2.0; x = sqrt(x))
   cout << "x es ahora igual a "  << x << endl;
```

5.3. PRECAUCIONES EN EL USO DE FOR

Un bucle `for` se debe construir con gran precaución, asegurándose de que la expresión de inicialización, la condición del bucle y la expresión de incremento harán que la condición del bucle se convierta en falsa en algún momento. En particular: «*si el cuerpo de un bucle de conteo modifica los valores de cualquier variable implicada en la condición del bucle, entonces el número de repeticiones se puede modificar*».

Esta regla anterior es importante, ya que su aplicación se considera una mala práctica de programación. Es decir, no es recomendable modificar el valor de cualquier variable de la condición del bucle

dentro del cuerpo de un bucle `for`, ya que se pueden producir resultados imprevistos. Por ejemplo, la ejecución de

```
int limite = 1;
for (int i = 0; i <= limite; i++)
{
   cout << i << endl;
   limite++;
}
```

produce una secuencia infinita de enteros (puede terminar si el compilador tiene constantes `MAXINT`, con máximos valores enteros, entonces la ejecución terminará cuando `i` sea `MAXINT` y `limite` sea `MAXINT+1 = MININT`).

```
0
1
2
3
.
.
.
```

ya que a cada iteración, la expresión `limite++` incrementa `limite` en `1`, antes de que `i++` incremente `i`. A consecuencia de ello, la condición del bucle `i <= limite` siempre es cierta.

Otro ejemplo es el bucle

```
int limite = 1;
for (int i = 0; i <= limite; i++)
{
   cout << i << endl;
   i--;
}
```

que producirá infinitos ceros

```
0
0
0
.
.
```

ya que en este caso la expresión `i--` del cuerpo del bucle decrementa `i` en `1` antes de que se incremente la expresión `i++` de la cabecera del bucle en `1`. Como resultado `i` es siempre `0` cuando el bucle se comprueba.

5.3.1. Bucles infinitos

El uso principal de un bucle `for` es implementar bucles de conteo en el que el número de repeticiones se conoce por anticipado. Por ejemplo, la suma de enteros de 1 a n. Sin embargo, existen muchos problemas en los que el número de repeticiones no se pueden determinar por anticipado. Para estas situa-

ciones algunos lenguajes tienen sentencias específicas tales como las sentencias `LOOP` de Modula-2 y Modula-3, el bucle `DO` de FORTRAN 90 o el bucle `loop` del de Ada. C++ no soporta una sentencia que realice esa tarea, pero existe una variante de la sintasix de `for` que permite implementar **bucles infinitos**, que son aquellos bucles que, en principio, no tienen fin.

Sintaxis

```
for  (;;)
  sentencia
```

`sentencia` se ejecuta indefinidamente a menos que se utilice una sentencia `return` o `break` (normalmente una combinación `if-break` o `if-return`).

La razón de que el bucle se ejecute indefinidamente es que se ha eliminado la expresión de inicialización, la condición del bucle y la expresión de incremento; al no existir una condición de bucle que especifique cuál es la condición para terminar la repetición de sentencias, éstas se ejecutarán indefinidamente. Así, el bucle

```
for (;;)
   cout << "Siempre así, te llamamos siempre así...";
```

producirá la salida

```
Siempre así, te llamamos siempre así...
Siempre así, te llamamos siempre así...
...
```

un número ilimitado de veces, a menos que el usuario interrumpa la ejecución (normalmente pulsando las teclas `Ctrl` y `C` en ambientes PC).

Para evitar esta situación, se requiere el diseño del bucle `for` de la forma siguiente:

1. El cuerpo del bucle ha de contener todas las sentencias que se desean ejecutar repetidamente.
2. Una sentencia terminará la ejecución del bucle cuando se cumpla una determinada condición.

La sentencia de terminación suele ser `if-break` con la sintaxis

```
if  (condición)  break;
```

`condición` *es una expresión lógica*
`break` *termina la ejecucción del bucle y transfiere el control a la sentencia siguiente al bucle*

y la sintaxis completa

```
for (;;)          // bucle
{
      lista_sentencias1
      if (condición_terminación) break;
      lista_sentencias2
}                 // fin del bucle
```

`lista_sentencias` *puede ser vacía, simple o compuesta*

Ejemplo 5.5

```
for (;;)
{
   cout << "Introduzca un número";
   cin >> num;
   if (num = -999) break;
   ...
}
```

5.3.2. Los bucles `for` vacíos

Tenga cuidado de situar un punto y coma después del paréntesis inicial del bucle `for`. Es decir, el bucle

```
for (int i = 1; i<= 10; i++); <-------- problema
   cout << "Sierra Magina" << endl;
```

no se ejecuta correctamente, ni se visualiza la frase `"Sierra Magina"` diez veces como era de esperar, ni se produce un mensaje de error por parte del compilador.

En realidad, lo que sucede es que se visualiza una vez la frase `"Sierra Magina"` ya que la sentencia `for` es una sentencia vacía al terminar con un punto y coma (;). Entonces, lo que sucede es que la sentencia `for` no hace absolutamente nada y tras 10 iteraciones y, por tanto; después de que el bucle `for` se ha terminado, se ejecuta la siguiente sentencia `cout` y se escribe `"Sierra Magina"`.

> El bucle `for` con cuerpos vacíos puede tener algunas aplicaciones, especialmente cuando se requieren ralentizaciones o temporizaciones de tiempo.

5.3.3. Sentencias nulas en bucles `for`

Cualquiera o todas las sentencias de un bucle `for` pueden ser nulas. Para ejecutar esta acción se utiliza el punto y coma (;) para marcar la sentencia vacía. Si se desea crear un bucle `for` que actúe exactamente como un bucle `while`, se deben incluir las primeras y terceras sentencias vacías.

```
// Listado
// bucles for con sentencias nulas

#include <iostream>
using namespace std;
int main()
{
   int contador = 0;

   for (; contador < 5;)
   {
      contador++;
      cout << "¡Bucle! ";
   }
```

```
16:       cout << "\n Contador: " << contador << " \n";
17:       return 0;
18:   }
```

Salida

```
¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle! ¡Bucle!
Contador: 5
```

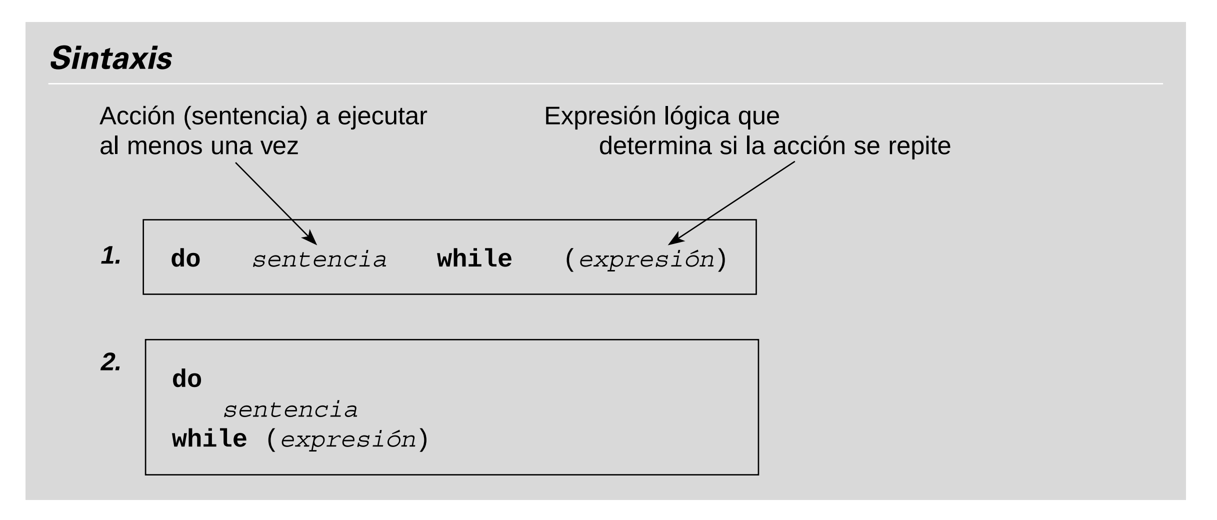
Análisis

En la línea 8 se inicializa la variable del contador. La sentencia `for` en la línea 10 no inicializa ningún valor, pero incluye una prueba de `contador < 5`. No existe ninguna sentencia de incrementación, de modo que el bucle se comporta exactamente como la sentencia siguiente.

```
while(contador < 5)
```

5.4. REPETICIÓN: EL BUCLE DO-WHILE

La sentencia **`do-while`** se utiliza para especificar un bucle condicional que se ejecuta al menos una vez. Esta situación se suele dar en algunas circunstancias en las que se ha de tener la seguridad de que una determinada acción se ejecutará una o varias veces, pero al menos una vez.



La construcción **do** comienza ejecutando *`sentencia`*. Se evalúa a continuación *`expresión`*. Si *`expresión`* es verdadera, entonces se repite la ejecución de sentencia. Este proceso continúa hasta que `expresión` es falsa. La semántica del bucle `do` se representa gráficamente en la Figura 5.3.

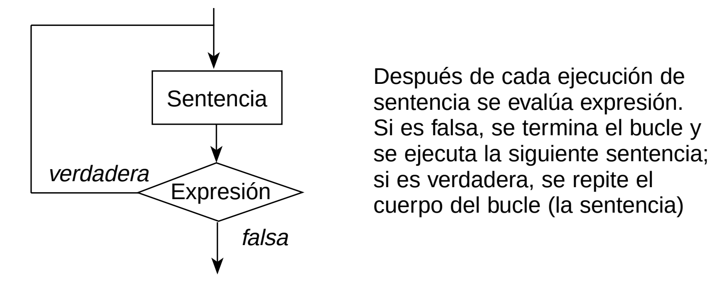


Figura 5.3. Diagrama de flujo de la sentencia **`do-while`**.

Ejemplo 5.6

```
do
{
   cout << "Introduzca un dígito (0-9): ";
   cin >> digito;
} while ((digito < '0') || ('9'< digito));
```

Este bucle se realiza mientras se introduzcan dígitos y se termina cuando se introduzca un carácter que no sea un dígito de `'0'` a `'9'`.

Ejercicio 5.2

Aplicación simple de un bucle `while`: seleccionar una opción de saludo al usuario dentro de un programa.

```
#include <iostream>
using namespace std;

int main()
{
   char opcion;
   do
   {
         cout << "Hola" << endl;
         cout << "¿Desea otro tipo de saludo?\n";
              << "Pulse s para si y n para no,\n"
              << "y a continuacion pulse intro: ";
         cin >> opcion;
   } while (opcion == 's'|| opcion == 'S');
   cout << "Adios\n";
   return 0;
}
```

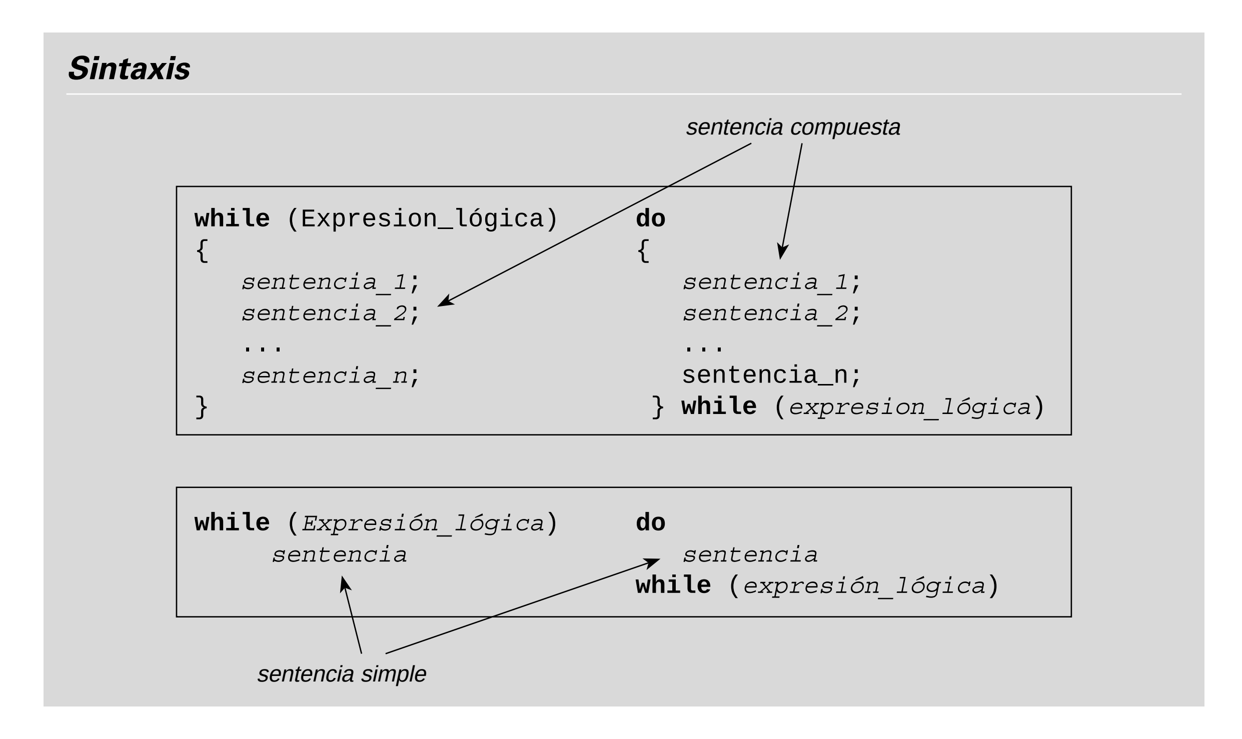
Salida de muestra

```
Hola
¿Desea otro tipo de saludo?
Pulse s para sí y n para no
y a continuación pulse intro: S
Hola
¿Desea otro tipo de saludo?
Pulse s para sí y n para no
y a continuación pulse intro: N
Adiós
```

5.4.1. Diferencias entre `while` y `do-while`

Una sentencia `do-while` es similar a una sentencia `while`, excepto que el cuerpo del bucle se ejecuta siempre al menos una vez.

Sintaxis



Ejemplo 1

```
// cuenta a 10
int x = 0;
do
   cout << "X:" << x++;
while (x < 10)
```

Ejemplo 2

```
// imprimir letras minúsculas del alfabeto
char car = 'a';
do
{
   cout << car << '';
   car++;
}while (car <= 'z');
```

Ejemplo 5.7

Visualizar las potencias de 2 cuerpos cuyos valores estén en el rango 1 a 1.000.

```
// ejercicio con while
potencia = 1;
while (potencia < 1000)
{
   cout << potencia << endl;
   potencia *= 2
} // fin de while
```

```
// ejercicio con do-while
potencia = 1;
do
{
   cout << potencia << endl;
      potencia *= 2;
} while (potencia < 1000);
```

5.5. COMPARACIÓN DE BUCLES `WHILE`, `FOR` Y `DO-WHILE`

C++ proporciona tres sentencias para el control de bucles: `while`, `for` y `do-while`. El bucle `while` se repite *mientras* su condición de repetición del bucle es verdadera; el bucle `for` se utiliza normalmente cuando el conteo esté implicado, o bien el control del bucle `for`, en donde el número de iteraciones requeridas se puede determinar al principio de la ejecución del bucle, o simplemente cuando existe una necesidad de seguir el número de veces que un suceso particular tiene lugar. El bucle `do-while` se ejecuta de un modo similar a `while` excepto que las sentencias del cuerpo del bucle se ejecutan siempre al menos una vez.

La Tabla 5.1 describe cuándo se usa cada uno de los tres bucles. En C++, el bucle `for` es el más frecuentemente utilizado de los tres. Es relativamente fácil reescribir un bucle `do-while` como un bucle `while`, insertando una asignación inicial de la variable condicional. Sin embargo, no todos los bucles `while` se pueden expresar de modo adecuado como bucles `do-while`, ya que un bucle `do-while` se ejecutará siempre al menos una vez y el bucle `while` puede no ejecutarse. Por esta razón, un bucle `while` suele preferirse a un bucle `do-while`, a menos que esté claro que se debe ejecutar una iteración como mínimo.

Tabla 5.1. Formatos de los bucles.

`while`	El uso más frecuente es cuando la repetición no está controlada por contador; el test de condición precede a cada repetición del bucle; el cuerpo del bucle puede no ser ejecutado. Se debe utilizar cuando se desea saltar el bucle si la condición es falsa.
`for`	Bucle de conteo cuando el número de repeticiones se conoce por anticipado y puede ser controlado por un contador; también es adecuado para bucles que implican control no contable del bucle con simples etapas de inicialización y de actualización; el test de la condición precede a la ejecución del cuerpo del bucle.
`do-while`	Es adecuada cuando se debe asegurar que al menos se ejecuta el bucle una vez.

Comparación de tres bucles

```
cuenta = valor_inicial;
while (cuenta < valor_parada)
{
   ...
   cuenta++;
} // fin de while

for(cuenta=valor_inicial; cuenta<valor_parada; cuenta++)
{
   ...
} // fin de for

cuenta = valor_inicial;
if (valor_inicial < valor_parada)
   do
   {
      ...
      cuenta++;
   } while (cuenta < valor_parada);
```

5.6. DISEÑO DE BUCLES

El diseño de un bucle requiere tres partes:

1. El cuerpo del bucle.
2. Las sentencias de inicialización.
3. Las condiciones para la terminación del bucle.

5.6.1. Bucles para diseño de sumas y productos

Muchas tareas frecuentes implican la lectura de una lista de números y calculan su suma. Si se conoce cuántos números habrá, tal tarea se puede ejecutar fácilmente por el siguiente pseudocódigo. El valor de la variable `total` es el número de números que se suman. La suma se acumula en la variable `suma`.

```
suma = 0;
repetir lo siguiente total veces:
   cin >> siguiente;
   suma = suma + siguiente;
fin_bucle
```

Este código se implementa fácilmente con un bucle `for`.

```
int suma = 0;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
   cin >> siguiente;
   suma = suma + siguiente;
}
```

Obsérvese que la variable `suma` se espera tome un valor cuando se ejecuta la siguiente sentencia

```
suma = suma + siguiente;
```

Dado que `suma` debe tener un valor la primera vez que la sentencia se ejecuta, `suma` debe estar inicializada a algún valor antes de que se ejecute el bucle. Con el objeto de determinar el valor correcto de inicialización de `suma` se debe pensar sobre qué sucede después de una iteración del bucle. Después de añadir el primer número, el valor de `suma` debe ser ese número. Esto es, la primera vez que se ejecute el bucle, el valor de `suma + siguiente` sea igual a `siguiente`. Para hacer esta operación *true* (verdadero), el valor de `suma` debe ser inicializado a 0.

Si en lugar de `suma`, se desea realizar productos de una lista de números, la técnica a utilizar es:

```
int producto = 1;
for (int cuenta = 1; cuenta <= total; cuenta++)
{
   cin << siguiente;
   producto = producto * siguiente;
}
```

La variable `producto` debe tener un valor inicial. No se debe suponer que todas las variables se deben inicializar a cero. Si `producto` se inicializara a cero, seguiría siendo cero después de que el bucle anterior se terminara.

5.6.2. Fin de un bucle

Existen cuatro métodos utilizados normalmente para terminar un bucle de entrada. Estos cuatro métodos son[1]:

1. Lista encabezada por tamaño.
2. Preguntar antes de la iteración.
3. Lista terminada con un valor centinela.
4. Agotamiento de la entrada.

Lista encabezada por el tamaño

Si su programa puede determinar el tamaño de una lista de entrada por anticipado, bien preguntando al usuario o por algún otro método, se puede utilizar un bucle «`repetir n veces`» para leer la entrada exactamente `n` veces, en donde `n` es el tamaño de la lista.

Preguntar antes de la iteración

El segundo método para la terminación de un bucle de entrada es preguntar, simplemente, al usuario, después de cada iteración del bucle, si el bucle debe ser o no iterado de nuevo. Por ejemplo:

```
suma = 0;
cout << "¿Existen números en la lista?:\n"
      << "teclee S para Sí, N para No y Final, Intro):";
char resp;
cin >> resp;
while ((resp == 'S')|| (resp == 'S'))
{
   cout << "Introduzca un número: ";
   cin >> número;
   suma = suma + numero;
   cout << "¿Existen más números?:\n";
         << "S para Sí, N para No. Final con Intro:";
   cin >> resp;
}
```

Este método es muy tedioso para listas grandes de números. Cuando se lea una lista larga es preferible incluir una única señal de parada, como se incluye en el método siguiente.

Valor centinela

El método más práctico y eficiente para terminar un bucle que lee una lista de valores del teclado es mediante un valor centinela. Un **valor centinela** es aquél que es totalmente distinto de todos los valores posibles de la lista que se está leyendo y de este modo sirve para indicar el final de la lista. Un ejemplo típico se presenta cuando se lee una lista de números positivos; un número negativo se puede utilizar como un valor centinela para indicar el final de la lista.

```
// ejemplo de valor centinela (número negativo)
...
```

[1] Estos métodos son descritos en Savitch, Walter, *Problem Solving with C++, The Object of Programming*, 2.ª edición, Reading, Massachussetts, Addison-Wesley, 1999.

```
cout << "Introduzca una lista de enteros positivos" << endl;
     << "Termine la lista con un número negativo" << endl;
suma = 0;
cin >> numero;
while (numero >= 0)
{
   suma = suma + numero;
   cin >> numero;
}
cout << "La suma es: " << suma;
```

Si al ejecutar el segmento de programa anterior se introduce la lista

```
4      8      15      -99
```

el valor de la suma será `27`. Es decir, `-99`, último número de la entrada de datos no se añade a suma. `-99` es el último dato de la lista que actúa como centinela y no forma parte de la lista de entrada de números.

Agotamiento de la entrada

Cuando se leen entradas de un archivo, se puede utilizar un valor centinela. Aunque el método más frecuente es comprobar simplemente si todas las entradas del archivo se han leído y se alcanza el final del bucle cuando no hay más entradas a leer. Éste es el método usual en la lectura de archivos, que suele utilizar una marca al final de archivo, `eof`. En el capítulo de archivos se dedicará una atención especial a la lectura de archivos con una marca de final de archivo.

5.6.3. OTRAS TÉCNICAS DE TERMINACIÓN DE BUCLE

Las técnicas más usuales para la terminación de bucles de cualquier tipo son:

1. Bucles controlados por contador.
2. Preguntar antes de iterar.
3. Salir con una condición bandera.

Un bucle **controlado por contador** es cualquier bucle que determina el número de iteraciones antes de que el bucle comience y, a continuación, repite (itera) el cuerpo del bucle esas iteraciones. La técnica de la lista encabezada por tamaño es un ejemplo de un bucle controlado por contador.

La técnica de **preguntar antes de iterar** se puede utilizar para bucles distintos de los bucles de entrada, pero el uso más común de esta técnica es para procesar la entrada.

La técnica del valor centinela es una técnica conocida también como **salida con una condición bandera** o **señalizadora**. Una variable que cambia su valor para indicar que algún suceso o evento ha tenido lugar, se denomina normalmente **bandera** o **indicador**. En el ejemplo anterior de suma de números, la variable bandera es `numero` de modo que cuando toma un valor negativo significa que indica que la lista de entrada ha terminado.

5.6.4. Bucles `for` vacíos

La sentencia nula `(;)` es una sentencia que está en el cuerpo del bucle y no hace nada.

```
// Listado
// Demostración de la sentencia nula
```

```
3:    // como cuerpo del bucle for
4:
5:    #include <iostream>
6:    using namespace std;
7:    int main()
8:    {
9:       for (int i = 0; i < 5; cout << "i: " << i++ << endl;
11:       ;
12:      return 0;
13:   }
```

Salida

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

Análisis

El bucle `for` de la línea 8 incluye tres sentencias: la sentencia de *inicialización* establece el contador `i` y se inicializa a `0`. La sentencia de *condición* comprueba `i < 5`, y la sentencia *`acción`* imprime el valor de `i` y lo incrementa.

5.6.5. Ruptura de control en bucles

En numerosas ocasiones, es conveniente disponer de la posibilidad de abandonar o salir de una iteración durante su ejecución, o dicho de otro modo, saltar sobre partes del código. El flujo de control en bucles se puede alterar de dos modos: insertar una sentencia `break` y una sentencia `continue`. La sentencia `break` termina el bucle; la sentencia `continue` termina la iteración actual del cuerpo del bucle. Además de estas sentencias, C++ dispone también de la sentencia `goto`, aunque este último caso no es muy recomendable usar.

> ***Sentencias para alteración del flujo de control***
>
> - `break` (ruptura).
> - `continue` (continuar).
> - `goto` (ir_a).

`break`

La sentencia `break` transfiere el control del programa al final del bucle (o en el caso de sentencia `swith` de selección, la sentencia más interna que lo encierra).

> ***Sintaxis***
>
> ```
> break;
> ```

Se puede utilizar para salir de un bucle infinito en lugar de comprobar el número en la expresión del bucle `while` o bucle `for`.

```
while (true)
{
   int a;

   cout << "a= ";
   cin >> a;
   if (!cin || a == b)
      break;
   cout << "número de la suerte: " << a << endl;
}
```

`continue`

La sentencia `continue` consta de la palabra reservada `continue` seguida por un punto y coma. Cuando se ejecuta, la sentencia `continue` termina la iteración actual del cuerpo del bucle actual más cercano que lo circunda.

Sintaxis

```
continue;
```

La sentencia provoca el retorno a la expresión de control del bucle `while`, o a la tercera expresión del bucle `for`, saltándose así el resto del cuerpo del bucle. Su uso es menos frecuente que `break`.

Uso de `continue`

```
while (expresión)
{
   sentencia1;
   if (car == '\n')
      continue;
   sentencia 2;
}
sentencia3;
```

Salta el resto del cuerpo del bucle y comienza una nueva iteración

Uso de `break`

```
while (cin.get (car))
{
   sentencia1;
   if (car == '\n')
      break;
   sentencia2;
}
sentencia n;
```

`break` *salta el resto del bucle y va a la sentencia siguiente*

Sentencia `goto`

C++, como otros lenguajes de programación, incluye la sentencia `goto` (ir_a), pero nunca se debe utilizar. Sólo en casos muy extremos se debe recurrir a ella.

Sintaxis

```
goto etiqueta;
goto excepción;
...
excepción;
```

Un caso extremo puede ser, salir de varios bucles anidados, ya que en este caso excepcional `break` sólo saca al bucle inmediatamente superior. Existen otros casos en que se requiere máxima eficiencia; por ejemplo, en un bucle interno de alguna solución en tiempo real o en el núcleo de un sistema operativo.

La sentencia `break` en bucles

La sentencia **`break`** se utiliza para realizar una terminación anormal del bucle. Dicho de otro modo, una terminación antes de lo previsto. Su sintaxis es:

```
break;
```

y se utiliza para salir de un bucle `while` o `do-while`, aunque también se puede utilizar dentro de una sentencia `switch`, siendo éste un uso muy frecuente:

```
while (condición)
{
    if (condición2)
       break;
    // sentencias
}
```

Ejemplo 5.8

El siguiente código extrae y visualiza valores de entrada desde el flujo de entrada `cin` hasta que se encuentra un valor especificado

```
int Clave;
cin >> Clave;
int Entrada;
while (cin >> Entrada) {
   if (Entrada != Clave)
      cout << Entrada << endl;
   else
      break;
}
```

Precaución

El uso de **break** en un bucle no es muy recomendable ya que puede hacer difícil la comprensión del comportamiento del programa. En particular, suele hacer muy difícil verificar los invariantes de los bucles. Por otra parte, suele ser fácil la reescritura de los bucles sin la sentencia **break**. El bucle anterior escrito sin la sentencia break.

```
int Clave;
cin >> Clave;
int Entrada;
while ((cin >> Entrada) && (Entrada != Clave))
{
   cout << Entrada << end1;
}
```

Sentencias break y continue

Las sentencias break y continue terminan la ejecución de un bucle while de modo similar a los otros bucles.

```
// Listado
// sentencias bucles for vacíos

#include <iostream>
using namospace std;

int main()
{
   int contador = 0;          // inicialización
   int max;
   cout << "Cuántos holas?";
   cin >> max;
   for (;;)         // bucle for que no termina nunca
   {
      if(contador < max)     // test
      {
         cout << "Hola!\n";
         contador++;         // incremento
      }
      else
        break;
   }
   return 0;
}
```

Salida

```
Cuántos holas? 3
Hola!
Hola!
Hola!
```

5.7. BUCLES ANIDADOS

Es posible *anidar* bucles. Los bucles anidados constan de un bucle externo con uno o más bucles internos. Cada vez que se repite el bucle externo, los bucles internos se repiten, se reevalúan los componentes de control y se ejecutan todas las iteraciones requeridas.

Ejemplo 5.9

El segmento de programa siguiente visualiza una tabla de multiplicación por cálculo y visualización de productos de la forma `x * y` para cada `x` en el rango de `1` a `Xultimo` y desde cada `y` en el rango `1` a `Yultimo` (donde `Xultimo`, y `Yultimo` son enteros prefijados). La tabla que se desea obtener es

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
        ...
```

```
for (int x = 1; x <= Xultimo; x++)

    for (int y = 1; y <= Yultimo; y++)
    {
       producto = x * y;
       cout << setw(2) << x << " * "
             << setw(2) << y << " = "
             << setw(3) << producto << endl;
    }
```

bucle externo *bucle interno*

El bucle que tiene x como variable de control se denomina **bucle externo** y el bucle que tiene y como variable de control se denomina **bucle interno**.

Ejemplo 5.10

```
// Aplicación de bucles anidados

#include <iostream>
#include <iomanip.h>      // necesario para cin y cout
using namespace std;      // necesario para setw

void main()
```

```
{
   // cabecera de impresión
   cout << setw(12) << " i " << setw(6) << " j " << endl;

   for (int i = 0; i < 4; i++)
   {
      cout << "Externo " << setw(7) << i << endl;
      for (int j = 0; j < i; j++)
         cout << "Interno " << setw(10) << j << endl;
   } // fin del bucle externo
}
```

La salida del programa es

```
              i    j
Externo       0
Externo       1
   Interno         0
Externo       2
   Interno         0
   Interno         1
Externo       3
   Interno         0
   Interno         1
   Interno         2
```

Ejercicio 5.3

Escribir un programa que visualice un triángulo isósceles.

```
                *
              * * *
            * * * * *
          * * * * * * *
        * * * * * * * * *
```

El triángulo isósceles se realiza mediante un bucle externo y dos bucles internos. Cada vez que se repite el bucle externo se ejecutan los dos bucles internos. El bucle externo se repite cinco veces (cinco filas); el número de repeticiones realizadas por los bucles internos se basan en el valor de la variable `fila`. El primer bucle interno visualiza los espacios en blanco no significativos; el segundo bucle interno visualiza uno o más asteriscos.

```
// archivo triángulo.cpp
#include <iostream>
using namespace std;

void main()
{
   // datos locales...
   const int num_lineas = 5;
   const char blanco = '';
   const char asterisco = '*';
```

```
    // comienzo de una nueva línea
    cout << endl;

    // dibujar cada línea: bucle externo
    for (int fila = 1; fila <= num_lineas; fila++)
    {
       // imprimir espacios en blanco: primer bucle interno
       for (int blancos = num_lineas –fila; blancos > 0;
          blancos--)
          cout << blanco;

       for (int cuenta_as = 1; cuenta_as < 2 * fila;
            cuenta_as ++)
            cout << asterisco;

       // terminar línea
       cout << endl;
    } // fin del bucle externo
 }
```

El bucle externo se repite cinco veces, uno por línea o fila; el número de repeticiones ejecutadas por los bucles internos se basa en el valor de `fila`. La primera fila consta de un asterisco y cuatro blancos, la fila 2 consta de tres blancos y tres asteriscos, y así sucesivamente; la fila 5 tendrá 9 asteriscos (2 ∂ 5 – 1).

Ejercicio 5.4

Ejecutar y visualizar el programa siguiente que imprime una tabla de m filas por n columnas y un carácter prefijado.

```
//Listado
//ilustra bucles for anidados

int main()
{
   int filas, columnas;
   char elCar;
   cout << "¿Cuántas filas?";
   cin >> filas;
   cout << "¿Cuántas columnas?";
   cin >> columnas;
   cout << "¿Qué carácter?";
   cin >> elCar;
   for (int i = 0; i < filas; i++)
   {
      for (int j = 0; j < columnas; j++)
         cout << elCar;
   cout << "\n";
   }
   return 0;
 }
```

Salida

```
¿Cuántas filas? 4
¿Cuántas columnas? 12
¿Qué carácter? *
************
************
************
************
```

Análisis

El usuario solicita el número de filas y columnas y un carácter a imprimir. El primer bucle `for` de la línea 14 inicializa un contador (`i`) a `0` y, a continuación, se ejecuta el cuerpo del bucle `for` externo.

En la línea 16 se inicializa otro bucle `for` y un segundo contador `j` se inicializa a `0` y se ejecuta el cuerpo del bucle interno. En la línea 17 se imprime el carácter `elCar` (un `*`, en la ejecucción). Se evalúa la condición (`j < columnas`) y si se evalúa a *true* (*verdadero*), `j` se incrementa y se imprime el siguiente carácter. Esta acción se repite hasta que `j` sea igual al número de columnas.

El bucle interno imprime 12 caracteres asterisco en una misma fila y el bucle externo repite cuatro veces (número de filas) la fila de caracteres.

RESUMEN

Un bucle es un grupo de instrucciones a la computadora que se ejecutan repetidamente hasta que una condición de terminación se cumple. Los bucles representan estructuras repetitivas y una estructura repetitiva es aquella que especifica que una acción (sentencia) se repite mientras que una condición especificada permanece verdadera (es cierta). Existen muchas formas de diseñar un bucle (lazo o ciclo) dentro de un programa C++, pero las más importantes son: repetición controlada por contador y repetición controlada por centinela.

- Un contador de bucle se utiliza para contar las repeticiones de un grupo de instrucciones. Se incrementa (o decrementa) normalmente en 1 cada vez que se ejecuta el grupo de instrucciones.
- Los valores centinela se utilizan para controlar la repetición cuando el número de repeticiones (iteraciones) no se conoce por adelantado y el bucle contiene sentencias que obtienen datos cada vez que se ejecuta. Un valor centinela se introduce después de que se han proporcionado todos los datos válidos al programa y debe ser distinto de los datos válidos.
- Los bucles `while` comprueban una condición, y si es verdadera (*true*), se ejecutan las sentencias del cuerpo del bucle.
- Los bucles `do-while` ejecutan el cuerpo del bucle y comprueban a continuación la condición.
- Los bucles `for` inicializan un valor y, a continuación, comprueban una expresión; si la expresión es verdadera se ejecutan las sentencias del cuerpo del bucle y se comprueba la expresión cada vez que termina la iteración. Cuando la expresión es falsa se termina el bucle y se sale del mismo.
- La sentencia `break` produce la salida inmediata del bucle.
- La sentencia `continue` cuando se ejecuta salta todas las sentencias que vienen a continuación y prosigue con la siguiente iteración del bucle.
- La sentencia `switch` maneja una serie de decisiones en las que se comprueban los valores de una variable o expresión determinada y en función de su valor realiza una acción diferente.

EJERCICIOS

5.1. Seleccione y escriba el bucle adecuado que mejor resuelva las siguientes tareas:

a) Suma de series tales como 1/2 + 1/3 + 1/4 + + 1/5 + ... + 1/50
b) Lectura de la lista de calificaciones de un examen de Historia
c) Visualizar la suma de enteros en el intervalo 11...50.

5.2. Considérese el siguiente código de programa

```
for (i = 0; i < n; ++i) {
     --n;
}
cout << i << endl;
```

a) ¿Cuál es la salida si `n` es 0?
b) ¿Cuál es la salida si `n` es 1?
c) ¿Cuál es la salida si `n` es 3?

5.3. ¿Cuál es la salida de los siguientes bucles?

```
int n, m;
for ( n = 1; n <= 10; n++)
      for (m = 10; m >= 1; m--)
             cout << n << "veces" << m
                       << " = " << n *
                       m << endl;
```

5.4. Escriba un programa que calcule y visualice

1 + 2 + 3 + ... + (n-1) + n

donde *n* es un valor de un dato.

5.5. ¿Cuál es la salida del siguiente bucle?

```
suma = 0;
while (suma < 100)
     suma += 5;
cout << suma << endl;
```

5.6. Escribir un bucle `while` que visualice todas las potencias de un entero `n`, menores que un valor especificado, `max_limite`.

5.7. ¿Qué hace el siguiente bucle `while`? Reescribirlo con sentencias `for` y `do-while`.

```
num = 10;
while (num <= 100)
{
     cout << num << endl;
     num += 10;
}
```

5.8. ¿Cuál es la salida de los siguientes bucles?

a)
```
for (int i = 0; i < 10; i++)
         cout << " 2* " << i << "
            = " << 2 * i << endl;
```

b)
```
for (int i = 0; i <= 5; i++)
         cout << 2 * i + 1 << " ";
cout << endl;
```

c)
```
for (int i = 1; i < 4; i++)
{
     cout << i;
     for (int j = i; j >= 1; j--)
        cout << j << endl;
}
```

5.9. Escribir un programa que visualice el siguiente dibujo:

```
        *
      * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
  * * * * * * *
    * * * * *
      * * *
        *
```

5.10. Describir la salida de los siguientes bucles:

a)
```
for (int i = 1; i <= 5; i++)
{
   cout << i << endl;
   for (int j = i; j >= 1; j-=2)
   cout << j << endl;
}
```

b)
```
for (int i = 3; i > 0; i--)
   for (int j = 1; j <= i; j++)
    for (int k = i; k >= j; k--)
        cout << i << j << k <<
endl;
```

c)
```
for (int i = 1; i <= 3; i++)
    for (int j = 1; j <= 3; j++)
    {
       for (int k = i; k <= j;
               k++)
       cout << i << j << k <<
             endl;
       cout << endl;
    }
```

5.11. Escribir un programa que lea una línea de texto y, a continuación, visualice el último carácter leído. Así, para la entrada

`Sierra de Cazorla`

la salida será la letra `a`.

5.12. Escribir un programa que calcule la suma de los números enteros comprendidos entre 1 y 50.

5.13. Escribir un programa que calcule y visualice una tabla con las 20 potencias de 2.

5.14. Imprimir los cuadrados de los enteros de 1 a 20.

5.15. Dada una serie de números enteros leídos de un archivo de entrada, calcular el factorial de cada uno de ellos.

PROYECTOS DE PROGRAMACIÓN

5.1. Diseñar e implementar un programa que cuente el número de sus entradas que son positivos, negativos y cero.

5.2. Diseñar e implementar un programa que extraiga valores del flujo de entrada estándar y, a continuación, visualice el mayor y el menor de esos valores en el flujo de salida estándar. El programa debe visualizar mensajes de advertencias cuando no haya entradas.

5.3. Diseñar e implementar un programa que solicite al usuario una entrada como un dato tipo fecha y, a continuación, visualice el número del día correspondiente del año. Ejemplo, si la fecha es 30 12 1999, el número visualizado es 364.

5.4. Un carácter es un espacio en blanco si es un blanco (`''`), una tabulación (`'\t'`), un carácter de nueva línea (`'\n'`) o un avance de página (`'\f'`). Diseñar y construir un programa que cuente el número de espacios en blanco de la entrada de datos.

5.5. Escribir un programa que lea una temperatura en grados Celsius e imprima el equivalente en grados Fahrenheit.

5.6. Escribir un programa que convierta: (*a*) centímetros a pulgadas; (*b*) libras a kilogramos.

5.7. Escribir un programa que lea el radio de una esfera y visualice su área y su volumen ($A = 4\sigma r^2$, $V = 4/3\sigma r^3$).

5.8. Escribir un programa que lea tres enteros positivos, `día, mes` y `anyo,` y, a continuación, visualice la fecha que represente, el número de días, del mes y una frase que diga si el año es o no bisiesto. Ejemplo, 4/11/1999 debe visualizar 4 de noviembre de 1999. Ampliar el programa de modo que calcule la fecha correspondiente a 100 días más tarde.

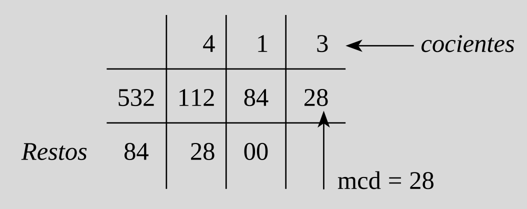
5.9. Escribir y ejecutar un programa que invierta los dígitos de un entero positivo dado.

5.10. Implementar el algoritmo de *Euclides* que encuentra el máximo común divisor de dos números enteros y positivos.

Algoritmo de Euclides de m y n

El algoritmo transforma un par de enteros positivos (*m*, *n*) en un par (*d*, *o*), dividiendo repetidamente el entero mayor por el menor y reemplazando el mayor con el resto. Cuando el resto es 0, el otro entero de la pareja será el máximo común divisor de la pareja original.

Ejemplo mcd (532, 112)



PROBLEMAS

5.1. En una empresa de computadoras, los salarios de los empleados se van a aumentar según su contrato actual:

Contrato	*Aumento %*
0 a 9.000 dólares	20
9.001 a 15.000 dólares	10
15.001 a 20.000 dólares	5
Más de 20.000 dólares	0

Escribir un programa que solicite el salario actual del empleado y calcule y visualice el nuevo salario.

5.2. La constante *pi* (3.1441592...) es muy utilizada en matemáticas. Un método sencillo de calcular su valor es:

$$Pi = 4 * \left(\frac{2}{3}\right) * \left(\frac{4}{5}\right) * \left(\frac{6}{5}\right) * \left(\frac{6}{7}\right) ...$$

Escribir un programa que efectúe este cálculo con un número de términos especificados por el usuario.

5.3. Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de N números. El valor de N se solicitará al principio del programa y los números serán introducidos por el usuario.

5.4. Escribir un programa que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (1988), excepto los múltiplos de 100 que no son bisiestos salvo que a su vez también sean múltiplos de 400 (1800 no es bisiesto, 2000 sí).

5.5. Escribir un programa que visualice un cuadrado mágico de orden impar *n*, comprendido entre 3 y 11; el usuario elige el valor de *n*. Un cuadrado mágico se compone de números enteros comprendidos entre 1 y n^2. La suma de los números que figuran en cada línea, cada columna y cada diagonal son idénticos. Un ejemplo es:

8	1	6
3	5	7
4	9	2

Un método de construcción del cuadrado consiste en situar el número 1 en el centro de la primera línea, el número siguiente en la casilla situada encima y a la derecha, y así sucesivamente. Es preciso considerar que el cuadrado se cierra sobre sí mismo: la línea encima de la primera es de hecho la última y la columna a la derecha de la última es la primera. Sin embargo, cuando la posición del número caiga en una casilla ocupada, se elige la casilla situada debajo del número que acaba de ser situado.

5.6. El matemático italiano Leonardo Fibonacci propuso el siguiente problema. Suponiendo que un par de conejos tiene un par de crías cada mes y cada nueva pareja se hace fértil a la edad de un mes. Si se dispone de una pareja fértil y ninguno de los conejos muertos, ¿cuántas parejas habrá después de un año? Mejorar el problema calculando el número de meses necesarios para producir un número dado de parejas de conejos.

5.7. Calcular la suma de la serie 1/1 + 1/2 + ... + 1/*N*, donde *N* es un número que se introduce por teclado.

5.8. Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de N números. El valor de N se solicitará al principio del programa y los números serán introducidos por el usuario.

5.9. Calcular el factorial de un número entero leído desde el teclado utilizando las sentencias `while`, `repeat` y `for`.

5.10. Encontrar el número mayor de una serie de números.

5.11. Calcular la media de las notas introducidas por teclado con un diálogo interactivo semejante al siguiente:

¿Cuántas notas? 20
Nota 1 : 7.50
Nota 2: 6.40
Nota 3: 4.20
Nota 4: 8.50
...
Nota 20: 9.50
Media de estas 20: 7.475

5.12. Escribir un programa que calcule la suma de los 50 primeros números enteros.

5.13. Calcular la suma de una serie de números leídos del teclado.

5.14. Contar el número de enteros negativos introducidos en una línea.

5.15. Visualizar en pantalla una figura similar a la siguiente

```
*
**
***
****
*****
```

siendo variable el número de líneas que se pueden introducir.

5.16. Encontrar el número natural N más pequeño tal que la suma de los N primeros números exceda de una cantidad introducida por el teclado.

5.17. Calcular todos los números de tres cifras tales que la suma de los cubos de las cifras es igual al valor del número.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 91).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

5.1. *¿Cuál es la salida del siguiente segmento de programa?*

```
for (int cuenta = 1; cuenta < 5;
     cuenta++)
       cout << (2 * cuenta) << " ";
```

5.2. *¿Cuál es la salida de los siguientes bucles?*

a)
```
for (int n = 10; n > 0; n = n – 2)
{
     cout << "Hola ";
     cout << n  << endl ;
}
```
b)
```
for (double n = 2; n > 0; n = n
     - 0.5)
     cout << m << " ";
```

5.3. *Considerar el siguiente código de programa.*

```
using namespace std;

int main(int argc, char *argv[])
{
    int i = 1 , n ;
    cin >> n;
    while (i <= n)
      if ((i % n) == 0)
            ++i;
    cout << i << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

a) ¿Cuál es la salida si se introduce como valor de n, 0?

b) ¿Cuál es la salida si se introduce como valor de n, 1?

c) ¿Cuál es la salida si se introduce como valor de n, 3?

5.4. *Suponiendo que m = 3 y n = 5 ¿Cuál es la salida de los siguientes segmentos de programa?*

a)
```
for (int i = 0; i < n; i++)
{
      for (int j = 0; j < i; j++)
            cout << " *";
      cout << endl;
}
```
b)
```
for (int i = n; i > 0; i--)
{
      for (int j = m; j > 0; j--)
            cout << " * ";
      cout << endl;
}
```

5.5. *¿Cuál es la salida de este bucle?*

```
int i = 1 ;
while (i * i < 10)
{
      int j = i;
      while (j * j < 100)
      {
            cout << i + j << " ";
            j *= 2;
      }
      i++;
      cout << endl;
}
cout << "\n*****\n";
```

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 91).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

5.1. Escriba un programa que calcule y visualice 1 + 2 + 3 + ... + (n-1) + n, donde n es un valor de un dato positivo.

5.2. Escribir un programa que visualice la siguiente salida:

```
1
1   2
1   2   3
1   2   3   4
1   2   3
1   2
1
```

5.3. Diseñar e implementar un programa que lea un total de 10 números y cuente el número de sus entradas que son positivos, negativos y cero.

5.4. Diseñar e implementar un programa que solicite a su usuario un valor no negativo n y visualice la siguiente salida (n = 6):

```
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

5.5. Escribir un programa que lea un límite máximo entero positivo, una base entera positiva, y visualice todas las potencias de la base, menores que el valor especificado límite máximo.

5.6. Diseñar un algoritmo que sume los m = 30 primeros números pares.

5.7. Escribir un programa que lea el radio de una esfera y visualice su área y su volumen.

5.8. Escribir un programa que presente los valores de la función coseno(3x)-2x para los valores de x igual a 0, 0.5, 1.0, ... 4.5, 5.

5.9. Escribir y ejecutar un programa que invierta los dígitos de un entero positivo dado leído del teclado.

5.10. Implementar el algoritmo de Euclides que encuentra el máximo común divisor de dos números enteros y positivos.

5.11. Escribir un programa que calcule y visualice el más grande, el más pequeño y la media de n números (n > 0).

5.12. Encontrar un número natural n más pequeño tal que la suma de los n primeros términos de la serie $\sum_{i=1}^{n} i * i - i - 2$ exceda de una cantidad introducida por el teclado máximo.

5.13. Calcular todos los números de exactamente tres cifras tales que la suma de los cuadrados de sus dígitos es igual al cociente de la división entera del número entre 3.

5.14. El valor de e^x se puede aproximar por la suma:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!} = \sum_{i=0}^{n} \frac{x^i}{i!}$$

Escribir un programa que lea un valor de x como entrada y visualice las sumas parciales de la serie anterior, cuando se ha realizado una suma, dos sumas, tres sumas, ..., 15 sumas.

5.15. Un número perfecto es un entero positivo, que es igual a la suma de todos los enteros positivos (excluido el mismo) que son divisores del número. El primer número perfecto es 6, ya que los divisores estrictos de 6 son 1, 2, 3 y 1 + 2 + 3 = 6. Escribir un programa que lea un número entero positivo tope y escriba todos los números perfectos menores o iguales que él.

5.16. Diseñar un programa que produzca la siguiente salida:

```
ZYXWVTSRQPONMLKJIHGFEDCBA
YXWVTSRQPONMLKJIHGFEDCBA
XWVTSRQPONMLKJIHGFEDCBA
WVTSRQPONMLKJIHGFEDCBA
VTSRQPONMLKJIHGFEDCBA
 ...
 ....
 .....
FEDCBA
EDCBA
DCBA
CBA
```

BA
A

5.17. Calcular la suma de la serie $1/1^3 + 1/2^3 + ... + + 1/n^3$ donde *n* es un número positivo que se introduce por teclado.

5.18. Calcular la suma de los 20 primeros términos de la serie: $1^2/3^2 + 2^2/3^2 + 3^2/3^3 + ... + n^2/3^n$.

5.19. Determinar si un número natural mayor o igual que la unidad que se lee del teclado es primo.

CAPÍTULO 6

Funciones

Contenido

INTRODUCCIÓN

Una función es un miniprograma dentro de un programa. Las funciones contienen varias sentencias bajo un solo nombre, que un programa puede utilizar una o más veces para ejecutar dichas sentencias. Las funciones ahorran espacio, reduciendo repeticiones y haciendo más fácil la programación, proporcionando un medio de dividir un proyecto grande en módulos pequeños más manejables. En otros lenguajes como BASIC o ensamblador se denominan *subrutinas;* en Pascal, las funciones son equivalentes a *funciones* y *procedimientos.*

Este capítulo examina el papel (rol) de las funciones en un programa C++. Las funciones pueden existir de modo autónomo o bien como miembros de una clase (Cap. 13). Como ya conoce, cada programa C++ tiene al menos una función **main();** sin embargo, cada programa C++ consta de muchas funciones en lugar de una función **main()** grande. La división del código en funciones hace que las mismas se puedan reutilizar en su programa y en otros programas. Después de que escriba, pruebe y depure su función, se puede utilizar nuevamente una y otra vez. Para reutilizar una función dentro de su programa, sólo se necesita llamar a la función.

Si se agrupan funciones en bibliotecas o como funciones miembro en bibliotecas de clases, otros programas se pueden reutilizar las funciones, por esa razón se puede ahorrar tiempo de desarrollo. Y dado que las bibliotecas contienen rutinas presumiblemente comprobadas, se incrementa la fiabilidad del programa completo.

La mayoría de los programadores no *construyen* bibliotecas, sino que, simplemente, las utilizan. Por ejemplo, cualquier compilador incluye más de quinientas funciones de biblioteca, que esencialmente pertenecen a la *biblioteca estándar* ANSI (*American National Standards Institute*). Dado que existen tantas funciones de bibliotecas, no siempre será fácil encontrar la función necesaria, más por la cantidad de funciones a consultar que por su contenido en sí. Por ello, es frecuente disponer del manual de biblioteca de funciones del compilador o algún libro que lo incluya.

La potencia real del lenguaje es proporcionada por la biblioteca de funciones. Por esta razón, será preciso conocer las pautas para localizar funciones de la biblioteca estándar y utilizarlas adecuadamente. En este capítulo aprenderá:

- Utilizar las funciones proporcionadas por la biblioteca estándar ANSI C, que incorporan todos los compiladores de C++.
- Los grupos de funciones relacionadas entre sí y los archivos de cabecera en que están declarados.

Las funciones son una de las piedras angulares de la programación en C++ y un buen uso de todas las propiedades básicas ya expuestas, así como de las propiedades avanzadas. Le proporcionarán una potencia, a veces impensable, a su programación. La compilación separada, la sobrecarga y la recursividad son propiedades cuyo conocimiento es esencial para un diseño eficiente de programas en numerosas aplicaciones.

CONCEPTOS CLAVE

- Ámbito (alcance).
- Argumento.
- Biblioteca de funciones.
- Compilación separada.
- Función.
- Parámetro referencia.
- Parámetro valor.
- Plantilla de funciones.
- Prototipo.
- Recursividad.
- Sobrecarga de funciones.
- Visibilidad.

6.1. CONCEPTO DE FUNCIÓN

C++ se puede utilizar como *lenguaje de programación estructurada,* también conocida como *programa-ciín modular*. Por esta razón, para escribir un programa se divide éste en varios módulos, en lugar de uno solo largo. El programa se divide en muchos módulos (rutinas pequeñas denominadas *funciones*), que producen muchos beneficios: aislar mejor los problemas, escribir programas correctos más rápido y producir programas que son más fáciles de mantener.

Así, pues, un programa C++ se compone de varias funciones, cada una de las cuales realiza una tarea principal. Por ejemplo, si está escribiendo un programa que obtenga una lista de caracteres del teclado, los ordene alfabéticamente y los visualice a continuación en la pantalla, se pueden escribir todas estas tareas en un único gran programa (función **`main()`**).

```
main()
{
   //Código C++ para obtener una lista de caracteres
   ...
   //Código C++ para alfabetizar los caracteres
   ...
   //Código C++ para visualizar la lista por orden alfabético
   ...
   return 0
}
```

Sin embargo, este método no es correcto. El mejor medio para escribir un programa es escribir funciones independientes para cada tarea que haga el programa. El mejor medio para escribir el citado programa sería el siguiente:

```
main()
{
   obtenercaracteres(); // Llamada a una función que obtiene los
                        // números
   alfabetizar();       // Llamada a la función que ordena
                        // alfabéticamente las letras
   verletras();         // Llamada a la función que visualiza
                        // letras en la pantalla
   return 0;            // retorno a DOS


int obtenercaracteres()
{
   //...
   // Código de C++ para obtener una lista de caracteres
   return 0;               //Retorno a main()
}

int alfabetizar()
{
   //...
   // Código de C++ para alfabetizar los caracteres
   //...
   return 0;               //Retorno a main()
}
```

```
int verletras()
{
   //...
   // Código de C++ para visualizar lista alfabetizada
   //...

   return 0;               //Retorno a main()
}
```

Cada función realiza una determinada tarea y cuando se ejecuta `return` se retorna al punto en que fue llamado por el programa o función principal.

> ***Consejo***
>
> Una buena regla para determinar la longitud de una función (número de líneas que contiene) es que no ocupe más longitud que el equivalente a una pantalla.

6.2. ESTRUCTURA DE UNA FUNCIÓN

Una función es, sencillamente, un conjunto de sentencias que se pueden llamar desde cualquier parte de un programa. Las funciones permiten al programador un grado de abstracción en la resolución de un problema.

Las funciones no se pueden anidar. Esto significa que una función no se puede declarar dentro de otra función. La razón para esto es permitir un acceso muy eficiente a los datos. En C++ todas las funciones son externas o globales, es decir, pueden ser llamadas desde cualquier punto del programa.

La estructura de una función en C++ se muestra en la Figura 6.1.

```
tipo_de_retorno nombreFunción (listaDeParámetros)
{
      cuerpo de la función
      return expresión
}
```

tipo_de_retorno	*tipo de valor devuelto por la función o la palabra reservada void si la función no devuelve ningún valor*
`nombreFunción`	*identificador o nombre de la función*
`listaDeParámetros`	*lista de declaraciones de los parámetros de la función separados por comas*
expresión	*valor que devuelve la función*

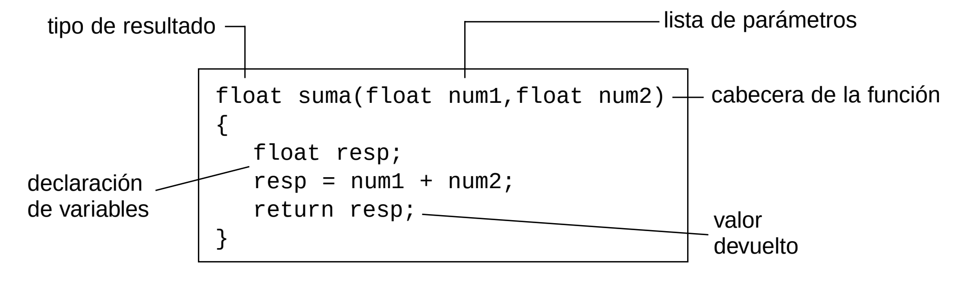


Figura 6.1. Estructura de una función.

Los aspectos más sobresalientes en el diseño de una función son:

- *Tipo de resultado.* Es el tipo de dato que devuelve la función C++ y aparece antes del nombre de la función.
- *Lista de parámetros.* Es una lista de parámetros *tipificados* (con tipos) que utilizan el formato siguiente:

  `tipo1 parámetro1, tipo2 parámetro2, ...`
- *Cuerpo de la función.* Se encierra entre llaves de apertura ({) y cierre (}). No hay punto y coma después de la llave de cierre.
- *Paso de parámetros.* Posteriormente se verá que el paso de parámetros en C++ se puede hacer por valor y por referencia.
- *No se pueden declarar funciones anidadas.*
- *Declaración local.* Las constantes, tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella.
- *Valor devuelto por la función.* Mediante la palabra reservada `return` se puede devolver el valor de la función.

Una llamada a la función produce la ejecución de las sentencias del cuerpo de la función y un retorno a la unidad de programa llamadora después que la ejecución de la función se ha terminado; normalmente, cuando se encuentra una sentencia `return`.

Ejemplo 6.1

Las funciones `cuadrado()` y `suma()` muestran dos ejemplos típicos de ellas.

```
// función que calcula los cuadrados de números enteros
// sucesivos a partir de un número dado (n), parámetro
// de la función, hasta obtener un cuadrado que sea
// mayor de 1000

int cuadrado(int n)
{
   int cuadrado = 0;
   while (cuadrado <= 1000)   //el cuadrado ha de ser menor de 1000
      cuadrado = n * n;
      cout   << "El cuadrado de: " << n << " es " << cuadrado
             << endl;
      n++;
   }
   return 0;
}

// Calcula la suma de un número dado (parámetro) de elementos de un
// array de datos

float suma (int num_elementos)
{
   int indice;
   float total = 0.0;
   for (indice = 0; indice <= num_elementos; indice++)
      total += Datos[indice]; //array Datos, cap. 7
   return total;
}
```

6.2.1. Nombre de una función

Un nombre de una función comienza con una letra o un subrayado (_) y puede contener tantas letras, números o subrayados como desee. El compilador —dependiendo del fabricante— ignora, sin embargo, a partir de una cantidad dada. C++ es sensible a mayúsculas, lo que significa que las letras mayúsculas y minúsculas son distintas a efectos del nombre de la función.

```
int max (int x, int y)              // nombre de la función max
double media (double x1, double x2) // nombre de la función media
```

6.2.2. Tipo de dato de retorno

Si la función no devuelve un valor `int`, se debe especificar el tipo de dato devuelto (de retorno) por la función. El tipo debe ser uno de los tipos simples de C++, tales como `int`, `char` o `float`, o un puntero a cualquier tipo C++, o un tipo `struct`.

```
int max(int x, int y)   // devuelve un tipo int
double media(double x1, double x2)   // devuelve un tipo double
float func0() {...}     //devuelve un float
char *func1() {...}     //devuelve un puntero a char
int  *func3() {...}     //devuelve un puntero a int
char *func4() {...}     //devuelve un puntero a un array char
int func5() {...}       //devuelve un int [es opcional]
```

Si una función no devuelve un resultado, se puede utilizar el tipo `void`, que se considera como un tipo de dato especial.

Algunas declaraciones de funciones que devuelven distintos tipos de resultados son:

```
int calculo_kilometraje(int litros, int kilometros);
char mayusculas(char car);
float DesvEst(void);
struct InfoPersona BuscarRegistro(int num_registro);
```

En C++, como se verá más adelante, el uso de la palabra reservada `struct` es opcional para funciones que devuelven un tipo estructura. Se puede utilizar solamente el nombre de la estructura, como en la última declaración:

```
InfoPersona BuscarRegistro(int num_registro);
```

Muchas funciones no devuelven resultados. La razón es que se utilizan como *subrutinas* para realizar una tarea concreta. Una función que no devuelve un resultado, a veces se denomina *procedimiento*. Para indicar al compilador que una función no devuelve resultado, se utiliza el tipo de retorno **`void`**, como en este ejemplo:

```
void VisualizarResultados(float Total, int num_elementos);
```

Si se omite un tipo de retorno para una función, como en

```
VerResultados(float Total, int longitud)
```

el compilador supone que el tipo de dato devuelto es `int`. Aunque el uso de `int` es opcional, por razones de claridad y consistencia se recomienda su uso. Así, la función anterior se puede declarar también:

```
int VerResultados(float Total, int longitud)
```

6.2.3. Resultados de una función

Una función puede devolver un único valor. El resultado se muestra con una sentencia `return` cuya sintaxis es:

```
return(expresión)

return(a + b + c);
return;
```

El valor devuelto (*expresión*) puede ser cualquier tipo de dato excepto una función o un *array*. Se pueden devolver valores múltiples devolviendo un puntero a una estructura o a un *array*. El valor de retorno debe seguir las mismas reglas que se aplican a un operador de asignación. Por ejemplo, no se puede devolver un valor `int` si el tipo de retorno es un puntero. Sin embargo, si se devuelve un `int` y el tipo de retorno es un `float`, se realiza la conversión automáticamente.

Una función puede tener cualquier número de sentencias `return`. Tan pronto como el programa encuentra cualquiera de las sentencias `return`, se retorna a la sentencia llamadora. La ejecución de una llamada a la función termina si no se encuentra ninguna sentencia `return`; en este caso, la ejecución continúa hasta la llave final del cuerpo de la función.

Si el tipo de retorno es `void`, la sentencia `return` se puede escribir como

```
return;
```

sin ninguna expresión de retorno, o bien, de modo alternativo, se puede omitir la sentencia `return`.

```
void func1(void)
{
   cout << "Esta función no devuelve valores";
}
```

El valor devuelto se suele encerrar entre paréntesis, pero su uso es opcional. En algunos sistemas operativos, como DOS, se puede devolver un resultado al entorno llamador. Normalmente, el valor 0 se suele devolver en estos casos.

```
int main()
{
   cout << "Hola mundo" << endl;
   return 0;
}
```

Consejo

Aunque no es obligatorio el uso de la sentencia `return` en la última línea, se recomienda su uso, ya que ayuda a recordar el retorno en ese punto a la función llamadora.

Precaución

Un error típico de programación es olvidar incluir la sentencia `return` o situarla dentro de una sección de código que no se ejecute. Si ninguna sentencia `return` se ejecuta, entonces el resultado que devuelve la función es impredecible y puede originar que su programa falle o produzca resultados incorrectos. Por ejemplo, suponga que se sitúa la sentencia `return` dentro de una sección de código que se ejecuta condicionalmente, tal como:

```
if (Total >= 0.0)
  return Total;
```

Si `Total` es menor que cero, no se ejecuta la sentencia `return` y el resultado de la función es un valor aleatorio. (C++ suele generar el mensaje de advertencia `"Function should return a value"`, que le ayudará a detectar este posible error.)

6.2.4. Llamada a una función

Las funciones, para poder ser ejecutadas, han de ser *llamadas* o *invocadas*. Cualquier expresión puede contener una *llamada a una función* que redirigirá el control del programa a la función nombrada. Normalmente la llamada a una función se realizará desde la función principal `main()`, aunque naturalmente también podrá ser desde otra función.

Nota

La función que llama a otra función se denomina *función llamadora* y la función controlada se denomina *función llamada*.

La función llamada que recibe el control del programa se ejecuta desde el principio y cuando termina (se alcanza la sentencia `return`, o la llave de cierre (`}`) si se omite `return`) el control del programa vuelve y retorna a la función `main()` o a la función llamadora si no es `main`.

```
void main()
{
   ...
   func1();
   func2();
   ...
   return;
}

void func1()
{
   ...
   return;
}

void func2()
{
   ...
return;
}
```

Figura 6.2. Traza de llamadas de funciones.

```
#include <iostream.h>

void func1(void)
{
   cout << "Segunda función \n";
   return;
}
void func2(void)
{
   cout << "Tercera función \n";
   return;
}

main()
{
   cout << "Primera función llamada main() \n";
   func1();                  //Segunda función llamada
   func2();                  //Tercera función llamada
   cout << "main se termina";

   return 0;                 //Se devuelve el control a DOS
}
```

La salida de este programa es:

```
Primera función llamada main()
Segunda función
Tercera función
main se termina
```

Se puede llamar a una función y no utilizar el valor que se devuelve. En el ejemplo

```
func();
```

el valor de retorno no se considera. El formato `func()` sin argumentos es el más simple. Para indicar que la llamada a una función no tiene argumentos se sitúa una palabra reservada `void` entre paréntesis en la declaración de la función y, posteriormente, en lo que se denominará *prototipo*.

```
void main()
{
   func();              //Llamada a la función
   ...
}
void func(void)         //Declaración de la función
{
   cout << "Hola mundo \n";
}
```

Precaución

No se puede definir una función dentro de otra. Todo código de la función debe ser listado secuencialmente a lo largo de todo el programa. Antes de que aparezca el código de una función, debe aparecer la llave de cierre de la función anterior.

Ejemplo 6.2

La función `max` *devuelve el número mayor de dos enteros.*

```
int max(int x, int y)
{
   if (x < y) return y;
   else
      return x;
}

void main()
{
   int m, n;
   do {
      cin >> m >> n;
      cout << max(m, n) << endl;      // llamada a max
   } while(m != 0);
}
```

Ejemplo 6.3

Calcular la media aritmética de dos números.

```
#include <iostream>
using namespace std;

double media(double x1, double x2)
{
   return(x1 + x2)/2;
}

void main()
{
   double num1, num2, med;
   cout << "Introducir dos números reales:";
   cin >> num1 >> num2;
   med = media(num1, num2);
   cout << "El valor medio es " << med << endl;
}
```

6.3. PROTOTIPOS DE LAS FUNCIONES

C++ requiere que una función se declare o defina antes de su uso. La *declaración* de una función se denomina *prototipo*. Los prototipos de una función contienen la misma cabecera de la función, con la diferencia de que los prototipos terminan con un punto y coma. Específicamente un prototipo consta de los siguientes elementos: nombre de la función, una lista de argumentos encerrados entre paréntesis y un punto y coma. La inclusión del nombre de los parámetros es opcional.

C++ requiere que esté declarada una función si se llama a esa función antes de que se defina.

Sintaxis

tipo_retorno nombre_función (lista_de_declaración_parámetros);

tipo_retorno	tipo del valor devuelto por la función o palabra reservada `void` si no devuelve un valor
nombre_función	nombre de la función
lista_declaración_parámetros	lista de declaración de los parámetros de la función, separados por comas (los nombres de los parámetros son opcionales, pero es buena práctica incluirlos para indicar los parámetros que representan)

Un prototipo declara una función y proporciona una información suficiente al compilador para verificar que la función está siendo llamada correctamente, con respecto al número y tipo de los parámetros y el tipo devuelto por la función. Es obligatorio poner un punto y coma al final del prototipo de la función con el objeto de convertirlo en una sentencia.

```
double FahrACelsius(double tempFahr);      // prototipo válido
int max(int, int);                         // prototipo válido
```

Los prototipos se sitúan normalmente al principio de un programa, antes de la definición de la primera función `main()`. El compilador utiliza los prototipos para validar que el número y los tipos de datos de los argumentos reales de la llamada a la función son los mismos que el número y tipo de argumentos formales en la función llamada. Si se detecta una inconsistencia, se visualiza un mensaje de error. Sin prototipos, un error puede ocurrir si un argumento con un tipo de dato incorrecto se pasa a una función. En programas complejos, este tipo de errores son difíciles de detectar.

En C++, la diferencia entre los conceptos *declaración* y *definición* es preciso tenerla clara. Cuando una entidad se *declara*, se proporciona un nombre y se listan sus características. Una *definición* proporciona un nombre de entidad y reserva espacio de memoria para esa entidad. Una *definición* indica que existe un lugar en un programa donde ´existe´ realmente la entidad definida, mientras que una *declaración* es sólo una indicación de que algo existe en alguna posición.

Una *declaración* de la función contiene sólo la cabecera de la función y una vez declarada la función, la *definición* completa de la función debe existir en algún lugar del programa; por ejemplo, antes o después de `main`.

```
int max(int, int);                     // declaración de max

// programa principal
void main()
{
   int m, n;
   do {
      cin >> m >> n;
      cout << max(m, n) << endl;
   } while (m != o);
}

// devuelve el entero mayor
int max(int x, int y)                  // definición de max
```

```
{
   if (x < y) return y;
   else return x;
}
#include <iostream.h>
double media (double x1, double x2);   // declaración de media

main()
{
   med = media (num1, num2);
   ...
}
double media(double x1, double x2)    // definición
{
   return (x1 + x2)/2;
}
```

Declaraciones de una función

- Antes de que una función pueda ser invocada, debe ser *declarada*.
- Una declaración de una función contiene sólo la cabecera de la función (llamado también *prototipo*).

```
tipo_resultado nombre (tipo1 param1, tipo2 param2, ...);
```

- Los nombres de los parámetros se pueden omitir.

```
double med (double, double);
double med (double x1, double x2);
```

La comprobación de tipos es una acción realizada por el compilador. El compilador conoce cuáles son los tipos de argumentos que se han pasado una vez que se ha procesado un prototipo. Cuando se encuentra una sentencia de llamada a una función, el compilador confirma que el tipo de argumento en la llamada a la función es el mismo tipo que el del argumento correspondiente del prototipo. Si no son los mismos, el compilador genera un mensaje de error. Un ejemplo de prototipo es

```
int procesar(int a, char b, float c, double d, char *e);
```

El compilador utiliza sólo la información de los tipos de datos. Los nombres de los argumentos no tienen significado y no se requieren; el propósito de los nombres es hacer la declaración de tipos más fácil para leer y escribir. La sentencia precedente se puede escribir también así:

```
int procesar(int, char, float, double, char *);
```

Si una función no tiene argumentos, se ha de utilizar la palabra reservada `void` como lista de argumentos del prototipo.

```
int muestra(void);
```

Ejemplos

```
1. // prototipo de la función cuadrado
   double cuadrado(double);
```

```
int main()
{
   cout << "5.6 al cuadrado =" << cuadrado(5.6) << endl;
   return 0;
}

double cuadrado(double n)
{
   return n * n;
}
```

2.
```
void visualizar_nombre(void);

void main()
{
   visualizar_nombre();
}

void visualizar_nombre()
{
   cout << "Hola Mackoy";
}
```

6.3.1. Prototipos con un número no especificado de parámetros

Un formato especial de prototipo es aquel que tiene un número no especificado de argumentos, que se representa por puntos suspensivos (...). Por ejemplo,

```
int muestras(int a, ...);
int printf(constante char *formato, ...);
int scanf(constante char *formato, ...);
```

6.4. PARÁMETROS DE UNA FUNCIÓN

C++ proporciona dos métodos para pasar variables (*parámetros*) entre funciones. Una función puede utilizar *parámetros por valor* y *parámetros por referencia*, o *puede no tener parámetros*. Esta sección examina el mecanismo que C++ utiliza para pasar parámetros a funciones y cómo optimizar el paso de parámetros, dependiendo del tipo de dato que se utiliza. Suponiendo que se tenga la declaración de una función **`círculo`** con tres argumentos

```
void circulo(int x, int y, int diametro);
```

Cuando se llama a `circulo` se deben pasar tres parámetros a esta función. En el punto de llamada cada parámetro puede ser una constante, una variable o una expresión, como en el siguiente ejemplo:

```
circulo(25, 40, vueltas*4);
```

6.4.1. Paso de parámetros por valor

Paso por valor (también llamado *paso por copia*) significa que cuando C++ compila la función y el código que llama a la función, la función recibe una copia de los valores de los parámetros. Si se cambia

el valor de un parámetro variable local, el cambio sólo afecta a la función y no tiene efecto fuera de la función.

La Figura 6.3 muestra la acción de pasar un argumento por valor. La variable real *i* no se pasa, pero el valor de *i*, 6, se pasa a la función receptora.

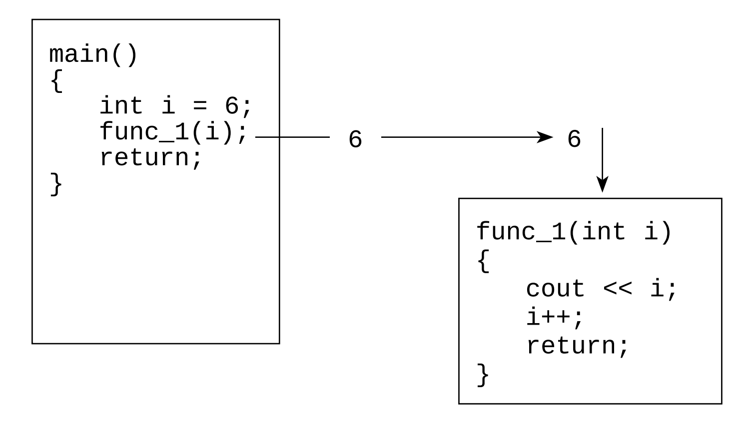


Figura 6.3. Paso de la variable *i* por valor.

En la técnica de paso de parámetro por valor, la función receptora no puede modificar la variable de la función (parámetro pasado).

Nota

El método por defecto de pasar parámetros es por valor, a menos que se pasen arrays. Los arrays se pasan siempre por dirección.

El listado `PARVALOR.UNO` muestra el mecanismo de paso de parámetros por valor.

```
// PARVALOR.UNO
// Muestra el paso de parámetros por valor
// Se puede cambiar la variable del parámetro en la función
// pero su modificación no puede salir al exterior

#include <iostream>
using namespace std;

void DemoLocal(int valor);

void main(void)
{
   int n = 10;
   cout << "Antes de llamar a DemoLocal, n = " << n << endl;
   DemoLocal(n);
   cout << "Después de llamada a DemoLocal, n " << n << endl;
   cout << "Pulse Intro (Enter) para continuar";
   cin.get();
}
```

```
void DemoLocal(int valor)
{
   cout << "Dentro de DemoLocal, valor = " << valor << endl;
   valor = 999;
   cout << "Dentro de DemoLocal, valor = " << valor << endl;
}
```

Al ejecutar este programa se visualiza la salida:

```
Antes de llamar a DemoLocal, n = 10
Dentro de DemoLocal, valor = 10
Dentro de DemoLocal, valor = 999
Después de llamar a DemoLocal, n = 10
```

6.4.2. Paso de parámetros por referencia

Cuando una función debe modificar el valor del parámetro pasado y devolver este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetro por *referencia* o *dirección*.

En este método el compilador pasa la dirección de memoria del valor del parámetro a la función. Cuando se modifica el valor del parámetro (la variable local), este valor queda almacenado en la misma dirección de memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado. Para declarar una variable parámetro como paso por referencia, el símbolo `&` debe preceder al nombre de la variable.

C++ permite utilizar punteros para implementar parámetros por referencia. Este método es el utilizado en C y se conserva en C++ precisamente por cuestión de compatibilidad.

```
// método de paso por referencia, mediante punteros
// estilo C

void intercambio(int* a, int* b)
{
   int aux = *a;
   *a      = *b;
   *b      = aux;
}
```

La función `intercambio()` utiliza las expresiones *`*a`* y *`*b`* para acceder a los enteros referenciados por las direcciones `int` de las variables *`i`* y *`j`* en la llamada de prueba siguiente:

```
int i = 3, j = 50;
cout << "i = " << i << " y j = " << j << endl;
intercambio(&i, &j);
cout << "i = " << i << " y j =" << j << endl;
```

La llamada a la función `intercambio()` debe pasar las direcciones de las variables intercambiadas. La versión de la función `intercambio()` que utiliza parámetros referencia es:

```
void intercambio(int& m, int& n)
{
   int aux = m;
   m       = n;
   n       = aux;
}
```

Los parámetros `m` y `n` son parámetros por referencia. Por consiguiente, cualquier cambio efectuado en el interior de la función se transmitirá al exterior de la misma. Para llamar a la función **`intercambio()`** por referencia, simplemente se pasan las variables a intercambiar, `i` y `j`.

```
int i = 3, j = 50;
cout << " i =" << i << " y j = " << j << endl;
intercambio(i, j);
cout << "i = " << i << " y j = " << j << endl;
```

6.4.3. Diferencia entre los parámetros por valor y por referencia

Las reglas que se han de seguir cuando se utilizan parámetros valor y referencia son las siguientes:

- los parámetros valor (declarados sin &) reciben copias de los valores de los argumentos que se les pasan;
- la asignación a parámetros valor de una función nunca cambian el valor del argumento original pasado a los parámetros;
- los parámetros referencia (declarados con &) reciben la dirección de los argumentos pasados;
- en una función, las asignaciones a parámetros referencia cambian los valores de los argumentos originales.

Por ejemplo, la escritura de una función `intercambio()` para intercambiar los contenidos de dos variables, requiere que los datos puedan ser modificados.

Llamada por valor	*Llamada por referencia*
`int uno, dos;`	`int uno, dos;`

```
intercambio1                  intercambio2
(int primero,int segundo)     (int& primero,int& segundo)
{                             {
...                             ...
}                             }
```

Sólo en el caso de `intercambio2` los valores de `uno` y `dos` se cambiarán. Veamos una aplicación completa de ambas funciones:

```
#include <iostream>
using namespace std;

void intercambio1(int, int);
void intercambio2(int&, int&);

void main()
{
   int a, b;

   a = 10; b = 20;

   intercambio1(a, b);
   cout << a << b << endl;

   intercambio2(a, b);
```

```
   cout << a << b << endl;
}

void intercambio1(int primero, int segundo)
{
   int aux;
   aux = primero;
   primero = segundo;
   segundo = aux;
}

void intercambio2(int& primero, int& segundo)
{
   int aux;
   aux = primero;
   primero = segundo;
   segundo = aux;
}
```

La ejecución del programa producirá:

```
10   20
20   10
```

Nota

Todos los parámetros en C se pasan por valor. C no tiene equivalente al parámetro referencia en C++.

6.4.4. Parámetros `const` de una función

Con el objeto de añadir seguridad adicional a las funciones, se puede añadir a una descripción de un parámetro el especificador `const`, que indica al compilador que sólo es de lectura en el interior de la función. Si se intenta escribir en este parámetro se producirá un mensaje de error de compilación.

```
void falso(const int item, const char& car)
{
   item = 123;            //Fallo en tiempo de compilación
   car = 'A'              //Fallo en tiempo de compilación
}
```

La Tabla 6.1 muestra un resumen del comportamiento de los diferentes tipos de parámetros.

Tabla 6.1. Paso de parámetros en C++.

Parámetro especificado como:	**Item pasado por**	**Cambia item dentro de la función**	**Modifica parámetros al exterior**
`int item`	*valor*	Sí	No
`const int item`	*valor*	No	No
`int& item`	*referencia*	Sí	Sí
`const int& item`	*referencia*	No	No

6.5. ARGUMENTOS POR OMISIÓN

Cuando una función tiene un cierto número de parámetros, normalmente el mismo número de argumentos deben indicarse cuando se llama a la función. En C++, sin embargo, es posible omitir algún argumento.

Una característica poderosa de las funciones C++ es que en ellas pueden establecer valores por *omisión* o *ausencia* («por defecto») para los parámetros. Se pueden asignar argumentos por defecto a los parámetros de una función. Cuando se omite el argumento de un parámetro que es un argumento por defecto, se utiliza automáticamente éste. La única restricción es que se deben incluir todas las variables desde la izquierda hasta el primer parámetro omitido. Si se pasan valores a los argumentos omitidos, se utiliza ese valor; si no se pasa un valor a un parámetro opcional, se utiliza el valor por defecto como argumento. El valor por defecto debe ser una expresión constante.

El ejemplo siguiente muestra cómo utilizar argumentos por defecto, en una función cuyo prototipo es:

```
void asteriscos(int fila, int col, int num, char c = '*');
```

Se puede llamar a la función **`asteriscos`** de dos formas equivalentes:

```
asteriscos(4, 0, 40);
```

o bien

```
asteriscos(4, 0, 40, '*');
```

Sin embargo, si se desea cambiar el carácter utilizado por la función, se puede escribir

```
asteriscos (4, 0, 40, '#')-;
```

en donde el argumento explícito (#) anula el carácter por omisión (`*`).

Otro ejemplo es la función `funcdef()`

```
char funcdef(int arg1=1, char c='A', float f_val=45.7f);
```

Se puede llamar a `funcdef` con cualquiera de las siguientes sentencias:

```
funcdef(9,'Z',91.5);  //Anula los tres argumentos por defecto
funcdef(25, 'W');     //Anula los dos primeros argumentos
funcdef(50);          //Anula el primer argumento
funcdef();            //Utiliza los tres argumentos por defecto
```

Las sentencias anteriores son equivalentes a:

```
funcdef(9, 'Z', 91.5);
funcdef(25, 'w', 45.7);
funcdef(50, 'A', 45.7);
funcdef(1, 'A', 45.7);
```

Sin embargo, no se puede omitir un argumento a menos que se omitan todos los argumentos a su derecha. Por ejemplo, la siguiente llamada a la función no es correcta:

```
funcdef( , 'Z', 99.99);
```

Se debe tener cuidado y situar cualquier argumento que tenga valores por defecto a la derecha de una función.

```
f()
f(a);
f(a, b);
f(a, b, c);
```

Ejemplo 6.4

La función `escribir_car` *tiene dos parámetros. El primero indica el carácter a escribir (visualizar) y el segundo indica el número de veces que ese carácter debe escribirse.*

```
void escribir_car(char c, int num = 1)
{
   for(int i = 1; i <= num; i++)
      cout << c;
}
```

Al parámetro `num` se le ha dado el valor por defecto de 1. Este valor se utilizará automáticamente si se omite el argumento correspondiente en una llamada. Si no se omite el argumento, se utilizará ese valor. Llamadas válidas son:

```
escribir_car('x', 4)      // se escribe cuatro 'x'
escribir_car('y');        // se escribe una 'y'
```

El programa `ARGDEFEC.CPP` muestra cómo asignar valores por omisión a variables parámetro.

```
// ARGDEFEC.CPP
// Muestra el uso de valores de parámetros por defecto

#include <iostream>
using namespace std;

void f(int a = 10, int b = 20, int c = 30)
{
   cout << "a = " << a << endl
        << "b = " << b << endl
        << "c = " << c << endl;
}

void main(void)
{
   f();
   f(1);
   f(1, 5);
   f(1, 2, 3);
   cout << "Pulse Intro (Enter) para continuar";
   cin.get();
}
```

Reglas de construcción de argumentos por defecto:

- Los argumentos por defecto se deben pasar por valor. Un nombre de un argumento por defecto no se puede pasar por referencia.
- Los valores de los argumentos por defecto pueden ser valores literales o definiciones `const`. *No pueden ser* variables. En otras palabras, si se declara `int n` y a continuación `int x = n`, se rechazará *n* como parámetro por defecto. Sin embargo, si *n* se declara con, `const int n=1` entonces se acepta la declaración.
- Todos los argumentos por defecto deben colocarse al final en el prototipo de la función. Después del primer argumento por defecto, todos los argumentos posteriores deben incluir también valores por defecto.

Ejemplo 6.5

Sea la función

```
void f(int anchura, float v = 3.14159, char x = '*');
```

Las llamadas siguientes son legales:

```
f(10);
f(10, 7.5);
f(10, 7.5, '$');
```

La llamada siguiente no es legal:

```
f(10, , '^');
```

ya que se ha intentado saltar uno de los parámetros por defecto.

Otro ejemplo puede ser la función `h()`:

```
void h(int a = 1, int b = 2, int c = 3, int d = 4);
```

y llamadas válidas son:

```
h();
h(9);
h(9, 18);
h(9, 18, 25); h(9, 18, 25, 4);
```

y llamadas no válidas:

```
h(9, , , 4);
```

Ejemplo

```
void escribir_car(char c, int num = 1, bool lineas_indep = false)
{
   for(int i = 1; i < num; i++)
```

```
{
   cout << c;
   if(lineas_indep)
      cout << endl;
}
}
```

Algunas llamadas válidas son

```
escribir_car('x');              // se escribe una x
escribie_car('#', 5);           // se escribe 5 * en la misma línea
escribir_car('%', 5, true);     // cinco % en líneas independientes
```

6.6. FUNCIONES EN LÍNEA (*INLINE*)

Cuando el compilador construye el código ejecutable de un programa C++ existen dos opciones disponibles para la generación del código de las funciones: *funciones en línea* y *fuera de línea*.

Las funciones en línea (`inline`) sirven para aumentar la velocidad de su programa. Su uso es conveniente cuando la función se utiliza muchas veces en el programa y su código es pequeño. Existe una diferencia grande en el comportamiento de ambos tipos de funciones.

Una función normal es un bloque de código que se llama desde otra función. El compilador genera código para situar la dirección de retorno en la pila. La dirección de retorno es la dirección de la sentencia que sigue a la instrucción que llama a la función. A continuación, el compilador genera códigos que sitúan cualquier argumento de la función en la pila a medida que se requiera. Por último, el compilador genera una instrucción de llamada que transfiere el control a la función.

Para una *función en línea* (`inline`), el compilador inserta realmente el código para la función en el punto en que se llama la función. Esta acción hace que el programa se ejecute más rápidamente, ya que no ha de ejecutar el código asociado con la llamada a la función.

Sin embargo, cada instancia de la función en línea puede requerir tanta memoria como se requiera para contener la función completa. Por esta razón, el programa incrementa su tamaño, aunque es mucho más rápido en su ejecución. Si se llama a una función en línea diez veces en un programa, el compilador inserta diez copias de la función en el programa. Si la función ocupa 1K, el tamaño de su programa se incrementa en 10K (10.240 bytes). Por el contrario, si se llama a la misma función con una función normal (no `inline`) diez veces, y el código de llamada suplementario es 25 bytes, la función requiere sólo 1.274 bytes (1K + 25 * 10, 1024 + 250, 1.274).

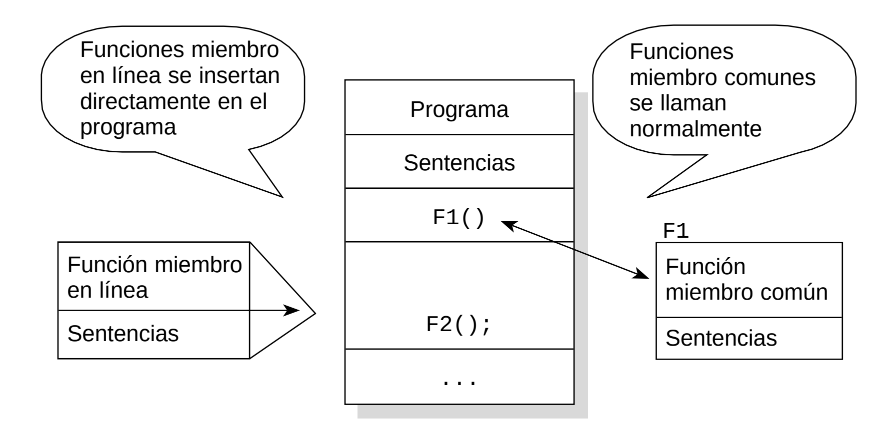


Figura 6.4. Código generado por una función fuera de línea.

La Figura 6.5 ilustra la sintaxis general de una función *inline*.

```
inline  TipoRetorno  NombreFunción (Lista parámetros con tipos)
```

Regla

Comience declarando las funciones *en-línea* como funciones ordinarias cuando desarrolle sus programas. Las funciones en línea son más fáciles de depurar. Una vez que su programa esté funcionando, inserte la palabra reservada `inline` donde sea necesario.

Figura 6.5. Código generado por una función en línea.

La Tabla 6.2 resume las ventajas y desventajas de situar un código de una función en línea o fuera de línea.

Tabla 6.2. Ventajas y desventajas de la función en línea.

	Ventajas	Desventajas
Funciones en línea	Rápida de ejecutar.	Tamaño de código grande.
Funciones fuera de línea	Pequeño tamaño de código.	Lenta de ejecución.

6.6.1. Creación de funciones en línea

Para crear una función en línea (**`inline`**), se debe insertar la palabra reservada `inline` delante de una declaración normal y del cuerpo, y situarla en el archivo fuente antes de que sea llamada. Por ejemplo, la función en línea `Sumar15` suma la constante 15 a un parámetro `n`:

```
inline int sumar15(int n) {return (n+15);}
```

La diferencia en la escritura reside en que el cuerpo de la función aparece inmediatamente después de la declaración. Las funciones en línea se suelen escribir en una sola línea dado su, casi siempre, pequeño tamaño, pero no existe ninguna restricción para declarar la función en varias líneas. Así, por ejemplo, la función `Sumar15` se podría haber escrito:

```
inline int Sumar15(int n)
{
   return (n + 15);
}
```

Ejercicio 6.1

Una aplicación completa de una función en línea `VolCono()`, que calcula el volumen de la figura geométrica `Cono`.

$$(V = \frac{1}{3} \pi r^2 h)$$

```
#include <iostream.h>

const float Pi = 3.141592;

inline float VolCono(float radio, float altura)
   {return ((Pi * (radio * radio) * altura) / 3.0);}

main()
{
   float radio, altura, volumen;
   cout << "Introduzca radio del cono";
   cin >> radio;
   cout << "Introduzca altura del cono";
   cin >> altura;
   volumen = cono(radio, altura);
   cout << "El volumen del cono es: " << volumen
   return 0;
}
```

6.7. ÁMBITO (ALCANCE)

El *ámbito* o *alcance* de una variable determina cuáles son las funciones que reconocen ciertas variables. Si una función reconoce una variable, la variable es *visible* en esa función. El ámbito es la zona de un programa en el que es visible una variable. Existen cuatro tipos de ámbitos: *programa*, *archivo fuente*, *función* y *bloque*. Se puede designar una variable para que esté asociada a uno de estos ámbitos. Tal variable es invisible fuera de su ámbito y sólo se puede acceder a ella en su ámbito.

Normalmente, la posición de la sentencia en el programa determina el ámbito. Los especificadores de clases de almacenamiento, `static`, `extern`, `auto` y `register`, pueden afectar al ámbito. El siguiente fragmento de programa ilustra cada tipo de ámbito:

```
int i;                 //Ámbito de programa
static int j;          //Ámbito de archivo
func(int k)            //Ámbito de programa, k ámbito de bloque
{
   int m;              //Ámbito de bloque
   ...                 //Ámbito de la función
}
```

6.7.1. Ámbito del programa

Las variables que tienen *ámbito de programa* pueden ser referenciadas por cualquier función en el programa completo; tales variables se llaman *variables globales*. Para hacer una variable global, declárela simplemente al principio de un programa, fuera de cualquier función.

```
int g, h;                  //variables globales
main()
{
   //...
}
```

Una variable global es visible («se conoce») desde su punto de definición en el archivo fuente. Es decir, si se define una variable global, cualquier línea del resto del programa, no importa cuántas funciones y líneas de código le sigan, podrá utilizar esa variable.

```
#include <iostream.h>
#include <iomanip.h>

float ventas, beneficios;      //variables globales
void f3(void){
   //...
}

void f1(void)
{
   //...
}
main()
{
   //...
}
```

Consejo

Declare todas las variables en la parte superior de su programa. Aunque se pueden definir tales variables entre dos funciones, podría realizar cualquier cambio en su programa de modo más rápido, si sitúa las variables globales al principio del programa.

6.7.2. Ámbito del archivo fuente

Una variable que se declara fuera de cualquier función y cuya declaración contiene la palabra reservada `static` tiene *ámbito de archivo fuente*. Las variables con este ámbito se pueden referenciar desde el punto del programa en que están declaradas hasta el final del archivo fuente. Si un archivo fuente tiene más de una función, todas las funciones que siguen a la declaración de la variable pueden referenciarla. En el ejemplo siguiente, *i* tiene ámbito de archivo fuente:

```
static int i;
void func(void)
{
   ...
}
```

6.7.3. Ámbito de una función

Una variable que tiene ámbito de una función se puede referenciar desde cualquier parte de la función. Las variables declaradas dentro del cuerpo de la función se dice que son *locales* a la función. Las variables locales no se pueden utilizar fuera del ámbito de la función en que están definidas.

6.7.4. Ámbito de bloque

Una variable declarada en un bloque tiene *ámbito de bloque* y puede ser referenciada en cualquier parte del bloque, desde el punto en que está declarada hasta el final del bloque. Las variables locales declara-

das dentro de una función tienen ámbito de bloque; no son visibles fuera del bloque. En el siguiente ejemplo, *i* es una variable local:

```
void funcdemo()
{
   int i;
   for (i = 0; i < 10; i++)
      cout << "i =" << i << "\n";
}
```

Una variable local declarada en un bloque anidado sólo es visible en el interior de ese bloque.

```
void func(int j)
{
   if (j > 3)
   {
      int i;
      for (i = 0; i < 50; i++)
          func2(i);
   }
   //aquí ya no es visible i
};
```

6.7.5. Variables locales

Además de tener un ámbito restringido, las variables locales son especiales por otra razón: existen en memoria sólo cuando la función está activa (es decir, mientras se ejecutan las sentencias de la función). Cuando la función no se está ejecutando, sus variables locales no ocupan espacio en memoria, ya que no existen. Algunas reglas que siguen las variables locales son:

- En el interior de una función, a menos que explícitamente cambie un valor de una variable, no se puede cambiar esa variable por ninguna sentencia externa a la función.
- Los nombres de las variables locales no son únicos. Dos o más funciones pueden definir la misma variable `test`. Cada variable es distinta y pertenece a su función específica.
- Las variables locales de las funciones no existen en memoria hasta que se ejecute la función. Por esta razón, múltiples funciones pueden compartir la misma memoria para sus variables locales (pero no al mismo tiempo).

6.8. CLASES DE ALMACENAMIENTO

Los especificadores de clases (tipos) de almacenamiento permiten modificar el ámbito de una variable. Los especificadores pueden ser uno de los siguientes: `auto`, `extern`, `register`, `static` y `typedef`.

6.8.1. Variables automáticas

Las variables que se declaran dentro de una función se dice que son automáticas (`auto`), significando que se les asigna espacio en memoria automáticamente a la entrada de la función y se les libera el espacio tan pronto se sale de dicha función. La palabra reservada `auto` es opcional.

```
auto int Total;        es igual que        int Total;
```

Normalmente no se especifica la palabra `auto`.

6.8.2. Variables externas

A veces, se presenta el problema de que una función necesita utilizar un valor de una variable global que *otra función* inicializa y se realiza compilación separada. Como las variables locales sólo existen temporalmente mientras se está ejecutando su función, no pueden resolver el problema. ¿Cómo se puede resolver entonces el problema? En esencia, de lo que se trata es de que una función de un archivo de código fuente utilice una variable definida en otro archivo. Una solución es declarar la variable local con la palabra reservada `extern`. Cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definida en otro lugar.

```
// exter1.cpp –– variables externas: parte 1

#include <iostream>
using namespace std;

void leerReal(void);

float f;

int main()
{
   leerReal();
   cout << "Valor de float =" << f;
   return 0;
}

// exter2.cpp –– variables externas: parte 2
#include <iostream>
using namespace std;

void leerReal(void)
{
   extern float f;

   cout << "Introduzca valor en coma flotante";
   cin >> f;
}
```

En el archivo `EXTER2.CPP` la declaración externa de *f* indica al compilador que *f* se ha definido en otra parte (archivo). Posteriormente, cuando estos archivos se enlacen, las declaraciones se combinan de modo que se referirán a las mismas posiciones de memoria.

6.8.3. Variables registro

Otro tipo de variable C++ es la *variable registro*. Precediendo a la declaración de una variable con la palabra reservada `register`, se sugiere al compilador que la variable se almacene en uno de los registros

hardware del microprocesador. La palabra `register` es una sugerencia al compilador y no una orden. La familia de microprocesadores 80x86 no tiene muchos registros hardware de reserva, por lo que el compilador puede decidir ignorar sus sugerencias.

Para declarar una variable registro, utilice una declaración similar a:

```
register int k;
```

Una variable registro debe ser local a una función, nunca puede ser global al programa completo.

El uso de la variable `register` no garantiza que un valor se almacene en un registro. Esto sólo sucederá si existe un registro disponible. Si no existen registros suficientes, C++ ignora la palabra reservada `register` y crea la variable localmente como ya se conoce.

Una aplicación típica de una variable registro es como variable de control de un bucle. Guardando la variable de control de un bucle en un registro, se reduce el tiempo que la CPU requiere para buscar el valor de la variable de la memoria. Por ejemplo,

```
register int indice;
for (indice = 0; indice < 1000; indice++)...
```

6.8.4. Variables estáticas

Las variables estáticas son opuestas, en su significado, a las variables automáticas. Las *variables estáticas* no se borran (no se pierde su valor) cuando la función termina y, en consecuencia, retienen sus valores entre llamadas a una función. Al contrario que las variables locales normales, una variable `static` se inicializa sólo una vez. Se declaran precediendo a la declaración de la variable con la palabra reservada `static`.

```
int func_uno()
{
   int i;
   static int j = 25;      //j, k variables estáticas
   static int k = 100:
}
```

Las variables estáticas se utilizan normalmente para mantener valores entre llamadas a funciones.

```
function ResultadosTotales(float Valor)
{
   static float Suma;
   ...
   Suma = Suma + Valor;
}
```

En la función anterior se utiliza Suma para acumular sumas a través de sucesivas llamadas a `ResultadosTotales`.

6.9. CONCEPTO Y USO DE FUNCIONES DE BIBLIOTECA

Todas las versiones del lenguaje C++ vienen con una biblioteca estándar de funciones en tiempo de ejecución que proporcionan soporte para operaciones utilizadas con más frecuencia. Estas funciones permiten realizar una operación con sólo una llamada a la función (sin necesidad de escribir su código fuente).

Las *funciones estándar* o *predefinidas*, como así se denominan las funciones pertenecientes a la biblioteca estándar, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo *archivo de cabecera*.

Los nombres de los archivos de cabecera estándar utilizados en nuestro programa se muestran a continuación encerrados entre corchetes tipo ángulo, con o sin la extensión `.h`:

```
<assert.h>    <ctype.h>    <errno.h>     <float.h>
<limits.h>    <math.h>     <setjmp.h>    <signal.h>
<stdarg.h>    <stdef.h>    <stdio.h>     <string.h>
<time.h>
```

En los módulos de programa se pueden incluir líneas `#include` con los archivos de cabecera correspondientes en cualquier orden, y estas líneas pueden aparecer más de una vez.

Para utilizar una función o un macro, se debe conocer su número de argumentos, sus tipos y el tipo de sus valores de retorno. Esta información se proporcionará en los prototipos de la función. La sentencia `#include` mezcla el archivo de cabecera en su programa.

Algunos de los grupos de funciones de biblioteca más usuales son:

- E/S estándar (para operaciones de Entrada/Salida);
- matemáticas (para operaciones matemáticas);
- rutinas estándar (para operaciones estándar de programas);
- visualizar ventana de texto;
- de conversión (rutinas de conversión de caracteres y cadenas);
- de diagnóstico (proporcionan rutinas de depuración incorporada);
- de manipulación de memoria;
- control del proceso;
- clasificación (ordenación);
- directorios;
- fecha y hora;
- de interfaz;
- diversas;
- búsqueda;
- manipulación de cadenas;
- gráficos.

Se pueden incluir tantos archivos de cabecera como sean necesarios en sus archivos de programa, incluyendo los suyos propios que definen sus funciones.

Muchas de estas funciones, tales como las rutinas de entrada/salida y las rutinas de memoria, no se suelen utilizar en programación C++. En su lugar se utilizan flujos (tales como `cin` y `cout`) y operadores (tales como `new` y `delete`). En este capítulo se estudiarán las funciones más sobresalientes y más utilizadas en programación, y en el Apéndice F (página web del libro) se detallan todos los archivos de cabecera y las funciones incluidas en esos archivos.

6.10. FUNCIONES DE CARÁCTER

El archivo de cabecera `<ctype.h>` define un grupo de funciones/macros de manipulación de caracteres. Todas las funciones devuelven un resultado de valor verdadero (distinto de cero) o falso (cero).

Para utilizar cualquiera de las funciones (Tabla 6.3) no se olvide incluir el archivo de cabecera `ctype.h` en la parte superior de cualquier programa que haga uso de esas funciones.

Tabla 6.3. Funciones de caracteres.

Función	Prueba (test) de
`int isalpha(int c)`	Letra mayúscula o minúscula.
`int isdigit(int c)`	Dígito decimal.
`int isupper(int c)`	Letra mayúscula (`A-Z`).
`int islower(int c)`	Letra minúscula (`a-z`).
`int isalnum(int c)`	`isalpha(c) || isdigit(c)`
`int iscntrl(int c)`	Carácter de control.
`int isxdigit(int c)`	Dígito hexadecimal.
`int isprint(int c)`	Carácter imprimible incluyendo ESPACIO.
`int isgraph(int c)`	Carácter imprimible excepto ESPACIO.
`int isspace(int c)`	`Espacio, avance de página, nueva línea, retorno de carro, tabulación, tabulación vertical.`
`int ispunct(int c)`	Carácter imprimible no espacio, dígito o letra.
`int toupper(int c)`	Convierte a letras mayúsculas.
`int tolower(int c)`	Convierte a letras minúsculas.

6.10.1. Comprobación alfabética y de dígitos

Existen varias funciones que sirven para comprobar condiciones alfabéticas:

- **`isalpha(c)`**
  Devuelve verdadero (distinto de cero) si `c` es una letra mayúscula o minúscula. Se devuelve un valor falso si se pasa un carácter distinto de letra a esta función.

- **`islower(c)`**
  Devuelve verdadero (distinto de cero) si `c` es una letra minúscula. Se devuelve un valor falso (0), si se pasa un carácter distinto de una minúscula.

- **`isupper(c)`**
  Devuelve verdadero (distinto de cero) si `c` es una letra mayúscula, falso con cualquier otro carácter.

Las siguientes funciones comprueban caracteres numéricos:

- **`isdigit(c)`**
  Comprueba si `c` es un dígito de 0 a 9, devolviendo verdadero (distinto de cero) en ese caso, y falso en caso contrario.

- **`isxdigit(c)`**
  Devuelve verdadero si `c` es cualquier dígito hexadecimal (`0` a `9`, `A` a `F`, o bien `a` a `f`) y falso en cualquier otro caso.

La siguiente función comprueba argumentos numéricos o alfabéticos:

- **`isalnum(c)`**
  Devuelve un valor verdadero si `c` es un dígito de 0 a 9 o un carácter alfabético (bien mayúscula o minúscula) y falso en cualquier otro caso.

Ejemplo 6.6

Leer un carácter del teclado y comprobar si es una letra.

```
// Archivo TESTMAMI.CPP
// Solicita iniciales y comprueba que son correctas

#include <iostream>
#include <ctype.h>

main()
{
   char inicial;

   cout << "¿Cuál es su primer carácter inicial?";
   cin >> inicial;
   while (isalpha(inicial))
   {
      cout << "\Carácter válido\n";
      cout << "¿Cuál es su siguiente inicial?";
      cin >> inicial;
   }
   cout << "¡Terminado!";
   return;
}
```

6.10.2. Funciones de prueba de caracteres especiales

Algunas funciones incorporadas a la biblioteca de funciones comprueban caracteres especiales, principalmente a efectos de legibilidad. Estas funciones son las siguientes:

- **`iscntrl(c)`**
  Devuelve verdadero si `c` es un *carácter de control* (códigos ASCII 0 a 31) y falso en caso contrario.

- **`isgraph(c)`**
  Devuelve verdadero si `c` es un carácter imprimible (no de control) excepto espacio; en caso contrario, se devuelve falso.

- **`isprint(c)`**
  Devuelve verdadero si `c` es un carácter imprimible, código ASCII 21 a 127, incluyendo un espacio; en caso contrario, se devuelve falso.

- **`ispunct(c)`**
  Devuelve verdadero si `c` es cualquier carácter de puntuación (un carácter imprimible distinto de espacio, letra o dígito); falso, en caso contrario.

- **`isspace(c)`**
  Devuelve verdadero si `c` es carácter un espacio, nueva línea (`\n`), retorno de carro (`\r`), tabulación (`\t`) o tabulación vertical (`\v`).

6.10.3. Funciones de conversión de caracteres

Existen funciones que sirven para cambiar caracteres mayúsculas a minúsculas o viceversa.

- **`tolower(c)`**
  Convierte el carácter `c` a minúscula, si ya no lo es.

- **`toupper(c)`**
  Convierte el carácter `c` a mayúscula, si ya no lo es.

Ejemplo 6.7

El programa `MAYMIN1.CPP` *comprueba si la entrada es una letra* `V` *o una letra* `H`*.*

```
// MAYMIN1.CPP
#include <iostream>
#include <ctype.h>
using namespace std;

int main()
{
   char resp;    //respuesta del usuario

   cout << "¿Es un varón o una hembra (V/H)?";
   cin.get(resp);
   resp=toupper(resp);
   switch (resp)
   {
      case 'V':
         cout << "Es un enfermero \n";
         break;
      case 'H':
         cout << "Es una maestra \n";
         break;
      default:
         cout << "No es ni enfermero ni maestra \n";
         break;
   }
   return 0;
}
```

6.11. FUNCIONES NUMÉRICAS

Virtualmente cualquier operación aritmética es posible en un programa C++. Las funciones matemáticas disponibles son las siguientes:

- matemáticas;
- trigonométricas;
- logarítmicas;
- exponenciales;
- aleatorias.

La mayoría de las funciones numéricas están en el archivo de cabecera `math.h`; las funciones **`abs`** y **`labs`** están definidas en `math.h` y `stdlib.h`, y las rutinas `div` y `ldiv` en `stdlib.h`.

6.11.1. Funciones matemáticas

Las funciones matemáticas usuales en la biblioteca estándar son:

- **`ceil(x)`**
  Redondea al entero más cercano.

- **`fabs(x)`**
  Devuelve el valor absoluto de `x` (un valor positivo).

- **`floor(x)`**
  Redondea por defecto al entero más próximo.

- **`fmod(x, y)`**
  Calcula el resto en coma flotante para la división *x/y*, de modo que $x = i*y+f$, donde *i* es un entero, *f* tiene el mismo signo que *x* y el valor absoluto de *f* es menor que el valor absoluto de *y*.

- **`pow(x, y)`**
  Calcula *x* elevado a la potencia *y* (x^y). Si *x* es menor que o igual a cero, *y* debe ser un entero. Si *x* es igual a cero, *y* no puede ser negativo.

- **`sqrt(x)`**
  Devuelve la raíz cuadrada de *x*. *x* debe ser mayor o igual a cero.

6.11.2. Funciones trigonométricas

La biblioteca de C++ incluye una serie de funciones que sirven para realizar cálculos trigonométricos. Es necesario incluir en su programa el archivo de cabecera `math.h` para utilizar cualquier función.

- **`acos(x)`**
  Calcula el arco coseno del argumento `x`. El argumento `x` debe estar entre `–1` y `1`.

- **`asin(x)`**
  Calcula el arco seno del argumento `x`. El argumento `x` debe estar entre `–1` y `1`.

- **`atan(x)`**
  Calcula el arco tangente del argumento `x`.

- **`atan2(x, y)`**
  Calcula el arco tangente de `x` dividido por `y`.

- **`cos(x)`**
  Calcula el coseno del ángulo `x` (`x` se expresa en radianes).

- **`sin(x)`**
  Calcula el seno del ángulo `x` (`x` se expresa en radianes).

- **`tan(x)`**
  Devuelve la tangente del ángulo `x` (`x` se expresa en radianes).

> ***Regla***
>
> Si necesita pasar un ángulo expresado en *grados* a radianes, para poder utilizarlo con las funciones trigonométricas, multiplique los grados por pi/180, donde pi = 3,14159.

6.11.3. Funciones logarítmicas y exponenciales

Las funciones logarítmicas y exponenciales suelen ser utilizadas con frecuencia no sólo en matemáticas, sino también en el mundo de la empresa y los negocios. Estas funciones requieren también el archivo de inclusión `math.h`.

- **`exp(x), expl(x)`**
  Calcula el exponencial e^x, donde `e` es la base de logaritmos naturales de valor 2.718282.

  ```
  valor = exp(5.0);
  ```

  Una variante de esta función es `expl`, que calcula e^x utilizando un valor `double long` (largo doble).

- **`log(x), logl(x)`**
  La función **`log`** calcula el logaritmo natural del argumento `x` y `logl(x)` calcula el citado logaritmo natural del argumento `x` de valor `long double` (largo doble).

- **`log10(x), log10l(x)`**
  Calcula el logaritmo decimal del argumento `x`, de valor real double en `log10(x)` y de valor real long double en `log10l(x)`. `x` ha de ser positivo.

6.11.4. Funciones aleatorias

Los números aleatorios son de gran utilidad en numerosas aplicaciones y requieren un trato especial en cualquier lenguaje de programación. C++ no es una excepción y la mayoría de los compiladores incorporan funciones que generan números aleatorios. Las funciones usuales de la biblioteca estándar de C++ son: **`rand`**, **`random`**, **`randomize`** y **`srand`**. Estas funciones se encuentran en el archivo `stdlib.h`.

- **`rand(void)`**
  La función **`rand`** genera un número aleatorio. El número calculado por **`rand`** varía en el rango de `0` a `RAND_MAX`. La constante `RAND_MAX` se define en el archivo `stdlib.h` como `2E15-1`. En consecuencia, asegúrese de incluir dicho archivo en la parte superior de su programa.

Cada vez que se llama a **`rand()`** en el mismo programa, se obtiene un número diferente. Sin embargo, si el programa se ejecuta una y otra vez, se devuelven el mismo conjunto de números aleatorios. Un método para obtener un conjunto diferente de números aleatorios es llamar a la función **`srand()`** o a la macro `randomize`.

La llamada a la función **`rand()`** se puede asignar a una variable o situar en una función de salida `cout` o `printf`.

```
test = rand();
cout << "Éste es un número aleatorio" << rand() << endl;
```

• **randomize(void)**
La macro `randomize` inicializa el generador de números aleatorios con una semilla aleatoria obtenida a partir de una llamada a la función **time**. Dado que esta macro llama a la función **time**, el archivo de cabecera `time.h` se incluirá en el programa. No se devuelve ningún valor.

```
// archivo aleato1.cpp

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
   clrscr();
   randomize();
   cout <<  "Número aleatorio" << random(10);

   return 0;
}
```

• **srand(semilla)**
La función **srand** inicializa el generador de números aleatorios. Se utiliza para fijar el punto de comienzo para la generación de series de números aleatorios; este valor se denomina `semilla`. Si el valor de `semilla` es 1, se reinicializa el generador de números aleatorios. Cuando se llama a la función **rand** antes de hacer una llamada a la función **srand**, se genera la misma secuencia que si se hubiese llamado a la función **srand** con el argumento `semilla` tomando el valor `1`.

• **random(num)**
La macro `random` genera un número aleatorio dentro de un rango especificado (`0` y el límite superior especificado por el argumento `num`). Se devuelve un número entre `0` y `num -1`.

```
// archivo aleato2.cpp. Compilador Borland C++

#include <iostream.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
   clrscr();
   randomize();
   cout << "Número aleatorio menor que 10 " << random(10);

   return 0;
}
```

Otro ejemplo de generación de números aleatorios, que fija la semilla en `50` es `aleato3.cpp`.

```
// archivo aleato3.cpp
#include <iostream.h>
#include <stdlib.h>
int main(void)
```

```
{
   clrscr();
   srand(50);
   cout << "Este es un número aleatorio " << rand();

   return 0;
}
```

6.12. FUNCIONES DE FECHA Y HORA

La familia de microprocesadores 80x86 tiene un sistema de reloj que se utiliza principalmente para controlar el microprocesador, pero se utiliza también para calcular la fecha y la hora.

El archivo de cabecera `time.h` define estructuras, macros y funciones para manipulación de fechas y horas. La fecha se guarda de acuerdo con el calendario gregoriano.

Las funciones time, **`clock, _strdate`** y **`_strtime`**, devuelven la hora actual como el número de segundos transcurridos desde la medianoche del 1 de enero de 1970 (hora universal, GMT), el tiempo de CPU empleado por el proceso invocante y la fecha y hora actual, respectivamente.

La estructura de tiempo también incluye los miembros siguientes:

```
struct tm
{
   int tm_sec;    /* segundos */
   int tm_min;    /* minutos */
   int tm_hour;   /* horas */
   int tm_mday;   /* día del mes 1 a 31 */
   int tm_mon;    /* mes, 0 para Ene, 1 para Feb,... */
   int tm_year;   /* año desde 1900 */
   int tm_wday;   /* días de la semana desde domingo (0-6) */
   int tm_yday;   /* día del año desde el 1 de Ene(0-365) */
   int tm_isdt;   /* siempre 0 para gmtime */
};
```

- **`clock(void)`**
  La función **`clock`** determina el tiempo de procesador transcurrido desde el principio de la ejecución del programa; es decir, determina el número de segundos. Si no se puede devolver el tiempo de procesador se devuelve –1.

  ```
  inicio = clock();
  fin    = clock();
  ```

- **`time(hora)`**
  La función **`time`** obtiene la hora actual; devuelve el número de segundos transcurridos desde la medianoche (00:00:00) del 1 de enero de 1970. Este valor de tiempo se almacena entonces en la posición apuntada por el argumento *hora*. Si *hora* es un puntero nulo, el valor no se almacena.
  El prototipo de la función es:

  ```
  time_t time(time_t *hora);
  ```

- **`localtime(hora)`**
  Convierte la fecha y hora en una estructura de tipo `tm`. Su prototipo es

  ```
  struct tm *localtime(const time_t *tptr);
  ```

- **`mktime(t)`**
  Convierte la hora a un formato de calendario. Toma la información de la hora local contenida en la estructura `*tptr` y determina los valores de todos los miembros en el formato de tiempo del calendario. Su prototipo es

```
time_t mktime(struct tm *tptr);

#include <dos.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
   struct tm tim;

   clrscr();
   tim.tm_year = 90;
   tim.tm_mon = 9;
   tim.tm_mday = 15;
   tim.tm_hour = 8;
   tim.tm_min = 45;
   tim.tm_sec = 30;
   tim.tm_isdt = 0;
   mktime (&tim);
   ...
   }
```

6.13. FUNCIONES DE UTILIDAD

C/C++ incluyen una serie de funciones de utilidad que se encuentran en el archivo de cabecera `stdlib.h` y que se listan a continuación.

- **`abs(n), labs(n)`**

```
int abs(int n);
long labs(long n);
```

  devuelven el valor absoluto de *n*.

- **`div(num, denom)`**

```
div_t div(int num, int denom);
```

  Calcula el cociente y el resto de `num`, dividido por `denom` y almacena el resultado en `quot` y `rem`, miembros `int` de la estructura `div_t`.

```
typedef struct
{
   int quot;      /* cociente */
   int rem;    /* resto */
} div_t;
```

  El siguiente ejemplo calcula y visualiza el cociente y el resto de la división de dos enteros.

```
#include <stdlib.h>
#include <stdio.h>
```

```
int main(void)
{
   div_t resultado;

   clrscr();
   resultado = div(16, 4);
   printf("Cociente", resultado.quot);
   printf("Resto", resultado.rem);
};
```

- **`ldiv(num, denom)`**
  Calcula el cociente y resto de `num` dividido por `denom`, y almacena los resultados de `quot` y `rem`, miembros `long` de la estructura `ldiv_t`.

```
typedef struct
{
   long int quot;      /* cociente */
   long int rem;          /* resto */
} ldiv_t;

resultado = ldiv(1600L, 40L);
```

6.14. VISIBILIDAD DE UNA FUNCIÓN

El *ámbito* de un objeto es su visibilidad desde otras partes del programa y la *duración* de un objeto es su tiempo de vida, lo que implica no sólo cuánto tiempo existe la variable, sino cuándo se crea y cuándo se hace disponible. El ámbito de un objeto en C++ depende de donde se sitúe la definición y de los modificadores que la acompañan. En resumen, se puede decir que un objeto definido dentro de una función tiene *ámbito local* (alcance local), o si se define fuera de cualquier función, se dice que tiene un *ámbito global*.

La Figura 6.6 resume el modo en que se ve afectado el ámbito por la posición en el archivo fuente.

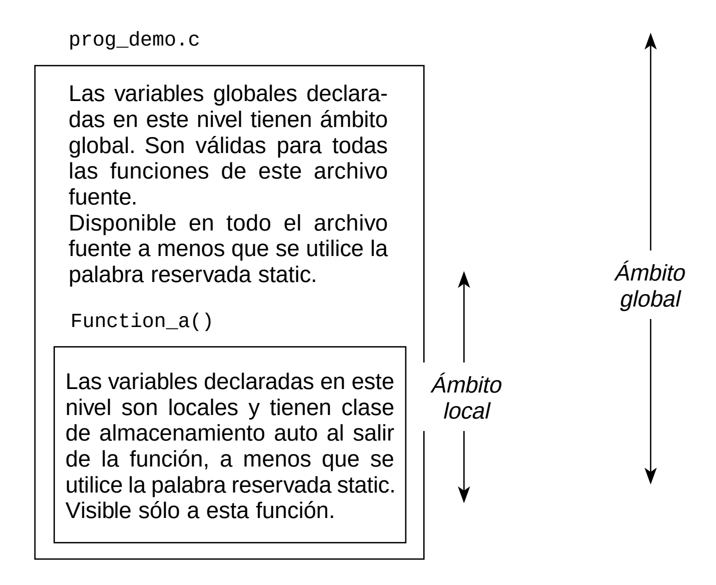


Figura 6.6. Ámbito de variable local y global.

Existen dos tipos de clases de almacenamiento en C++: `auto` y `static`. Una variable `auto` es aquella que tiene una *duración automática*. No existe cuando el programa comienza la ejecución, se crea en algún punto durante la ejecución y desaparece en algún punto antes de que el programa termine la ejecución. Una variable `static` es aquella que tiene una *duración fija*. El espacio para el objeto se establece en tiempo de compilación; existe en tiempo de ejecución y se elimina sólo cuando el programa desaparece de memoria en tiempo de ejecución.

Las variables con ámbito global se denominan *variables globales* y son las definidas externamente a la función (*declaración externa*). Las variables globales tienen el siguiente comportamiento y atributos:

- *Las variables globales tienen duración estática por defecto.* El almacenamiento se realiza en tiempo de compilación y nunca desaparece. Por definición, una variable global no puede ser una variable auto.
- *Las variables globales son visibles globalmente en el archivo fuente.* Se pueden referenciar por cualquier función, a continuación del punto de definición del objeto.
- *Las variables globales están disponibles, por defecto, a otros archivos fuente.* Esta operación se denomina *enlace externo*.

6.14.1. Variables locales frente a variables globales

Además de las variables globales, es preciso considerar las variables locales. Una *variable local* está definida solamente dentro del bloque o cuerpo de la función y no tiene significado (*vida*) fuera de la función respectiva. Por consiguiente, si una función define una variable como local, el ámbito de la variable está protegido. La variable no se puede utilizar, cambiar o borrar desde cualquier otra función sin una programación específica mediante el paso de valores (parámetros).

> Una ***variable local*** es una variable que se define dentro de una función.

> Una ***variable global*** es una variable que puede ser utilizada por todas las funciones de un programa dado, incluyendo `main()`.

Para construir variables globales en C++, se deben definir fuera de la función **`main()`**. Para ilustrar el uso de variables locales y globales, examine la estructura de bloques de la Figura 6.7. Aquí la variable

```
int x0;                  // variable global
funcion1(...)            // prototipo funcional

main
{
   ...
   ...
   ...
}

   funcion1(...)
   {
      int x1;            // variable local
      ...
      ...
      ...
   }
```

Figura 6.7. `x0` es global al programa completo, mientras que `x1` es local a la función `funcion1()`.

global es `x0` y la variable local es `x1`. La función puede realizar operaciones sobre `x0` y `x1`. Sin embargo, `main()` sólo puede operar con `x0`, ya que `x1` no está definida fuera del bloque de la función `funcion1()`. Cualquier intento de utilizar `x1` fuera de `funcion1()` producirá un error.

Examine ahora la Figura 6.8. Esta vez existen dos funciones, ambas definen `x1` como variable local. Nuevamente `x0` es una variable global. La variable `x1` sólo se puede utilizar dentro de las dos funciones. Sin embargo, cualquier operación sobre `x1` dentro de `funcion1()` no afecta al valor de `x1` en `funcion2()` y viceversa. En otras palabras, la variable `x1` de `funcion1()` se considera una variable independiente de `x1` en `funcion2()`.

Al contrario que las variables, *las funciones son externas por defecto*. Es preciso considerar la diferencia entre *definición* de una función y *declaración*. Si una declaración de variable comienza con la palabra reservada `extern`, no se considera definición de variable. Sin esta palabra reservada es una definición. Cada definición de variable es al mismo tiempo una declaración de variable. Se puede utilizar una variable sólo después de que ha sido declarada (en el mismo archivo). Únicamente las definiciones de variables asignan memoria y pueden, por consiguiente, contener inicializaciones. Una *variable sólo se define una vez*, pero se puede *declarar* tantas veces como se desee. Una declaración de variable al nivel global (externa a las funciones) es válida desde esa declaración hasta el final del archivo; una declaración en el interior de una función es válida sólo en esa función. En este punto, considérese que las definiciones y declaraciones de variables globales son similares a las funciones; la diferencia principal es que se puede escribir la palabra reservada `extern` en declaraciones de función.

```
int x0-;
funcion1()-;                        // prototipo funcion1
funcion2()                          // prototipo funcion2

main()
{
   ...
   ...
   ...
}

funcion1()
{
   int x1-;                   // variable local
   ...
   ...
   ...
}

funcion2()
{
   int x1;
   ...
   ...
   ...
}
```

Figura 6.8. `x0` es global al programa completo; `x1` es local tanto a `funcion1()` como a `funcion2()`, pero se tratan como variables independientes.

La palabra reservada `extern` se puede utilizar para notificar al compilador que la declaración del resto de la línea no está definida en el archivo fuente actual, pero está localizada en otra parte. El siguiente ejemplo utiliza `extern`:

```
// main.cpp
int Total;
extern int Suma;
extern void f(void);
void main(void)
...

// MODULO1.CPP
int Suma;
void f(void)
...
```

Utilizando la palabra reservada `extern` se puede acceder a símbolos externos definidos en otros módulos. `Suma` y la función `f()` se declaran externas.

> Las funciones son externas por defecto, al contrario que las variables.

6.14.2. Variables estáticas y automáticas

Los valores asignados a las variables locales de una función se destruyen cuando se termina la ejecución de la función y no se puede recuperar su valor para ejecuciones posteriores de la función. Las variables locales se denominan *variables automáticas*, significando que se pierden cuando termina la función. Se puede utilizar `auto` para declarar una variable

```
auto int ventas;
```

aunque las variables locales se declaran automáticas, por defecto y, por consiguiente, el uso de `auto` es opcional y, de hecho, no se utiliza.

Las *variables estáticas* (`static`), por otra parte, mantienen su valor después que una función se ha terminado. Una variable de una función, declarada como estática, mantiene un valor a través de ejecuciones posteriores de la misma función. Haciendo una variable local estática, su valor se retiene de una llamada a la siguiente de la función en que está definida. Se declaran las variables estáticas situando la palabra reservada `static` delante de la variable. Por ejemplo,

```
static int ventas = 10000;
static int dias = 500;
```

Este valor se almacena en la variable estática, sólo la primera vez que se ejecuta la función. Si su valor no está definido, el compilador almacena un cero en una variable estática por defecto.

El siguiente programa ilustra el concepto estático de una variable:

```
#include <iostream>
using namespace std;

// prototipo de la función
void Ejemplo_estatica(int);
```

```
void main()
{
   Ejemplo_estatica(1);
   Ejemplo_estatica(2);
   Ejemplo_estatica(3);
}
// Ejemplo del uso de una variable estática
void Ejemplo_estatica(int Llamada)
{
   static int Cuenta;
   if (Llamada == 1)
      Cuenta = 1;
   cout << "\n El valor de Cuenta en llamada nº " << Llamada
        << " es: " << Cuenta;
   ++Cuenta;
}
```

Al ejecutar el programa se visualiza:

```
El valor de Cuenta en llamada nº 1 es: 1
El valor de Cuenta en llamada nº 2 es: 2
El valor de Cuenta en llamada nº 3 es: 3
```

Si quita la palabra reservada `static` de la declaración de `Cuenta`, el resultado será:

```
El valor de Cuenta en llamada nº 1 es: 1
El valor de Cuenta en llamada nº 2 es: 1046
```

en donde no se puede predecir cuál es el valor de `Cuenta` en llamadas posteriores.

Las variables globales se pueden ocultar de otros archivos fuente utilizando el especificador de almacenamiento de clase `static`.

Para hacer una variable global privada al archivo fuente (y por consiguiente, no útil a otros módulos de código) se le hace preceder por la palabra `static`. Por ejemplo, las siguientes variables se declaran fuera de las funciones de un archivo fuente:

```
static int m = 25;
static char linea_texto[80];
static int indice_linea;
static char bufer[MAXLOGBUF];
static char *pBuffer;
```

Las variables anteriores son privadas al archivo fuente. Observe este ejemplo:

```
const OFF = 0
const ON = 1
...
static unsigned char maestro = OFF;
...

main()
{
   //...
```

```
}

funcion_a()
{
   //...
}
```

`maestro` se puede utilizar tanto en `funcion_a()` como en `main()`, en este archivo fuente, pero no se puede declarar como `extern` a otro módulo fuente.

Se puede hacer también una declaración de función `static`. Por defecto, todas las funciones tienen enlace externo y son visibles a otros módulos de programa. Cuando se sitúa la palabra reservada `static` delante de la declaración de la función, el compilador hace privada la función al módulo fuente. Se puede, entonces, reutilizar el nombre de la función en otros módulos fuente del programa.

6.15. COMPILACIÓN SEPARADA

Hasta este momento, casi todos los ejemplos que se han expuesto en el capítulo se encontraban en un solo archivo fuente. Los programas grandes son más fáciles de gestionar si se dividen en varios archivos fuente, también llamados *módulos*, cada uno de los cuales puede contener una o más funciones. Estos módulos se compilan y enlazan por separado posteriormente con un *enlazador*, o bien con la herramienta correspondiente del entorno de programación. Cuando se divide un programa grande en pequeños, los únicos archivos que se recompilan son los que se han modificado. El tiempo de compilación se reduce, dado que pequeños archivos fuente se compilan más rápido que los grandes. Los archivos grandes son difíciles de mantener y editar, ya que su impresión es un proceso lento que utilizará cantidades excesivas de papel.

La Figura 6.9 muestra cómo el enlazador puede construir un programa ejecutable, utilizando módulos objetos cada uno de los cuales se obtiene compilando un modelo fuente.

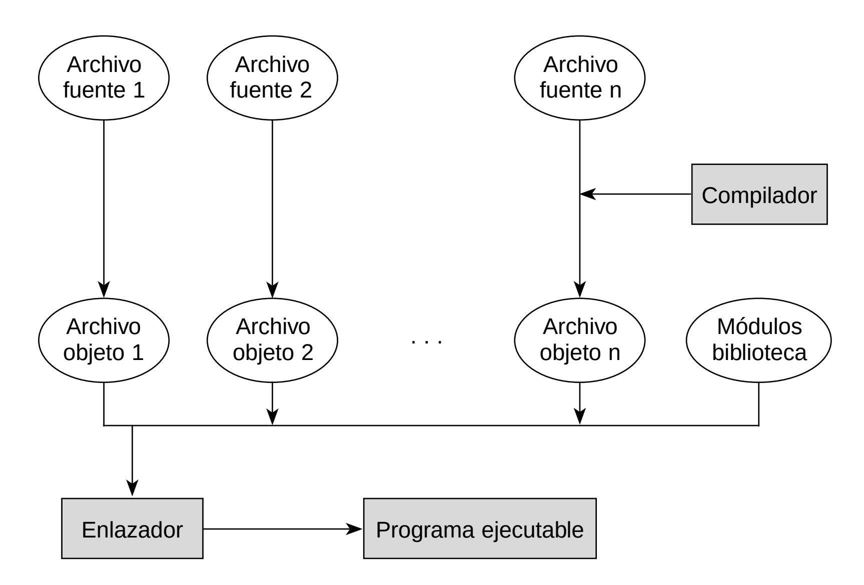


Figura 6.9. Compilación separada.

Cuando se tiene más de un archivo fuente, se puede referenciar una función en un archivo fuente desde una función de otro archivo fuente. Al contrario que las variables, las funciones son externas por

defecto. Si desea, por razones de legibilidad —no recomendable—, puede utilizar la palabra reservada con un prototipo de función y en una cabecera.

Se puede desear restringir la visibilidad de una función, haciéndola visible sólo a otras funciones en un archivo fuente. Una razón para hacer esto es reducir la posibilidad de tener dos funciones con el mismo nombre. Otra razón es reducir el número de referencias externas y aumentar la velocidad del proceso de enlace.

Se puede hacer una función no visible al exterior de un archivo fuente utilizando la palabra reservada `static` con la cabecera de la función y la sentencia del prototipo de función. Se escribe la palabra `static` antes del tipo de valor devuelto por la función. Tales funciones no serán públicas al enlazador, de modo que otros módulos no tendrán acceso a ellas. La palabra reservada `static`, tanto para variables globales como para funciones, es útil para evitar conflictos de nombres y prevenir el uso accidental de ellos. Por ejemplo, imaginemos un programa muy grande que consta de muchos módulos, en el que se busca un error producido por una variable global; si la variable es estática, se puede restringir su búsqueda al módulo en que está definida; si no es así, se extiende nuestra investigación a los restantes modelos en que está declarada (con la palabra reservada `extern`).

> Como regla general, son preferibles las variables locales a las globales. Si realmente es necesario o deseable que alguna variable sea global, es preferible hacerla estática, lo que significa que será «local» en relación al archivo en que está especificado.

Ejemplo 6.8

Supongamos dos módulos: `MODULO1` *y* `MODULO2`.

```
// MODULO1.CPP
#include <iostream>
using namespace std;

void main()
{
   void f(int i), g(void);
   extern int n;       // Declaración de n (no definición)
   f(8);
   n++;
   g();
   cout << "Fin de programa \n";
}

// MODULO2.CPP

#include <iostream>
using namespace std;

int n = 100;        // Definición de n (también declaración)

static int m = 7;

void f(int i)
{
   n += i + m;
```

```
}

void g(void)
{
   cout << "n =" << n << endl;
}
```

`f` y `g` *se definen* en el módulo 2 y *se declaran* en el módulo 1. Si se ejecuta el programa, se produce la salida

```
n = 116
Fin de programa.
```

> Se puede hacer una función invisible fuera de un archivo fuente utilizando la palabra reservada `static` con la cabecera y el prototipo de la función.

6.16. VARIABLES REGISTRO (*register*)

Una *variable registro* (`register`) es similar a una variable local, pero en lugar de ser almacenada en la pila, se almacena directamente en un registro del procesador (tal como *Si* o *bx*). Dado que el número de registros es limitado y además están limitados en tamaño, el número de variables registro que un programa puede crear simultáneamente es muy restringido.

Para declarar una variable registro, se hace preceder a la misma con la palabra reservada `register ;`

```
register int k;
```

La ventaja de las variables registro es su mayor rapidez de manipulación. Esto se debe a que las operaciones sobre valores situados en los registros son normalmente más rápidas que cuando se realizan sobre valores almacenados en memoria. Su uso se suele restringir a segmentos de código créditos. Las variables registro pueden ayudar a optimizar el rendimiento de un programa proporcionando acceso directo de la CPU a los valores claves del programa.

Una variable registro debe ser local a una función; nunca puede ser global al programa completo. El uso de la palabra reservada `register` no garantiza que un valor sea almacenado en un registro. Esto sólo sucederá si un registro está disponible (libre). Si no existen registros disponibles, C++ crea la variable como si fuera una variable local normal.

Una aplicación usual de las variables registro es como variable de control de bucles for o en la expresión condicional de una sentencia `while`, que se deben ejecutar a alta velocidad.

```
void usoregistro(void)
{
   register int k;

   cout << "\n Contar con una variable registro\n";
   for (k = 1; k <= 100; k++)
      cout << setw(8) << dec << k;
}
```

6.17. SOBRECARGA DE FUNCIONES (POLIMORFISMO)

C++ soporta una de las propiedades más sobresalientes en el mundo de la programación: *la sobrecarga*. La sobrecarga de funciones permite escribir y utilizar múltiples funciones con el mismo nombre, pero

con diferente lista de argumentos. La lista de argumentos es diferente si tiene un argumento con un tipo de dato distinto, si tiene un número diferente de argumentos, o ambos. La lista de argumentos se suele denominar *signatura de la función*.

Consideremos la función **`cuadrado`** definida de la forma siguiente:

```
int cuadrado(int x)
{
   return x * x;
}
```

Si se desea implementar una función similar para procesar un valor `long` o `double` y no se dispone de la propiedad de sobrecarga, se pueden definir funciones independientes que utilicen un nombre diferente para cada tipo, tal como:

```
long lcuadrado(long x);
double dcuadrado(double x);
```

Como C++ soporta sobrecarga se podrían definir funciones sobrecargadas que tuvieran el mismo nombre y argumentos diferentes:

```
int cuadrado(int x);
long cuadrado(long x);
double cuadrado(double x);
```

En su programa basta con llamar a la función `cuadrado()` con el argumento correcto:

```
long radio = 42500;
resultado = cuadrado(radio);
```

El compilador C++ verifica el tipo de parámetro enviado para determinar cuál es la función a llamar. Así, en la llamada anterior, C++ llama a

```
long cuadrado(long x);
```

ya que el parámetro radio es un tipo de dato `long`.

6.17.1. ¿Cómo determina C++ la función sobrecargada correcta?

Como se ha indicado anteriormente, C++ determina cuál de las funciones sobrecargadas ha de llamar, en función del número y tipo de parámetros pasados. C++ requiere que al menos uno de los parámetros tenga un tipo diferente del que utilizan otras funciones. Por consiguiente, es válido declarar dos funciones sobrecargadas, tales como

```
int f(int a, int b, int c);
int f(int a, int b, float c);
```

C++ puede llamar a ambas funciones, ya que al menos uno de los valores de los parámetros tiene un tipo diferente. Sin embargo, si se intenta escribir una función sobrecargada que tenga tipos similares, C++ emitirá un error. Por ejemplo, las definiciones siguientes no se compilarán, ya que el compilador no diferenciará unas de otras:

```
int f(int a, int b, int c);
int f(int a, int b, int c);
```

Las reglas que sigue C++ para seleccionar una función sobrecargada son:

- Si existe una correspondencia exacta entre los tipos de parámetros de la función llamadora y una función sobrecargada, se utiliza dicha función.
- Si no existe una correspondencia exacta, pero sí se produce la conversión de un tipo a un tipo superior (tal como un parámetro `int` a `long`, o un `float` a un `double`) y se produce, entonces, una correspondencia, se utilizará la función seleccionada.
- Se puede producir una correspondencia de tipos, realizando conversiones forzosas de tipos (*moldes-cast*).
- Si una función sobrecargada se define con un número variable de parámetros (mediante el uso de puntos suspensivos [...]), se puede utilizar como una coincidencia potencial.

Ejemplo

```
#include <iostream>
using namespace std;

// Prototipos de funciones
int Prueba(int);
int Prueba(int, int);
float Prueba(float, float);

void main()
{  int indice = 7;              // definición de variables
   int x = 4;   int y = 5;
   float a = 6.0;
   float b = 7.0; cout << "El cuadrado de" << índice << " es: " << Prueba(indice);
   cout << "\n El producto de " << x << "por" << y << " es: "
        << Prueba(x, y);
   cout << "\n La media de " << a << "y" << b << " es: "
        << Prueba(a, b);
}

// Prueba, calcula el cuadrado de un valor entero
int Prueba(int valor)
{
   return (valor * valor);
}

// Prueba, multiplica dos valores enteros
int Prueba(int valor1, int valor2)
{
   return(valor1 * valor2);
}

float prueba (float valor1, float valor2)
// Prueba, calcula la media de dos valores reales
{
   return ((valor1 + valor2) / 2);
}
```

Al ejecutar el programa se visualizará:

```
El cuadrado de 7 es: 49
El producto de 4 por 5 es: 20
La media de 6 y 7 es: 6.5
```

La sobrecarga de funciones es un tipo específico de polimorfismo. El **polimorfismo** es uno de los conceptos fundamentales de la programación orientada a objetos, como se verá más tarde.

El listado `POLIFUNC.CPP` muestra otro ejemplo completo de uso de funciones sobrecargadas, `seletipo`.

```
// POLIFUNC.CPP _ _ ilustra la sobrecarga de funciones
#include <iostream>
using namespace std;

void seletipo(int i);         // prototipos de la función
                              // sobrecargada
void seletipo(char *s);
void seletipo(float f);
void seletipo(char);
void seletipo(int i, int j);

void main()
{
   seletipo(5);
   seletipo("Cadena");
   seletipo(3.65f);
   seletipo('A');
   seletipo(4, 5);
}

void seletipo(int i)
{
   cout << i << "es un entero \n";
}

void seletipo(char *s)
{
   cout << s << "es una cadena \n";
}
void seletipo(float f)
{
   cout << f << " es un tipo float \n";
}

void seletipo(char c)
{
   cout << c << " es un tipo carácter \n";
}

void seletipo(int i, int j)
{
   cout << i << "y" << j << " son 2 enteros \n";
}
```

Los resultados de la ejecución de `POLIFUNC.CPP` son:

```
5 es un entero
Cadena es una cadena
3.65 es un tipo float
A es un tipo carácter
4 y 5 son 2 enteros
```

6.18. RECURSIVIDAD

Una *función recursiva* es una función que se llama a sí misma directa o indirectamente. La *recursividad* o *recursión directa* es el proceso por el que una función se llama a sí misma desde el propio cuerpo de la función. La *recursividad* o *recursión indirecta* implica más de una función.

La recursividad indirecta implica, por ejemplo, la existencia de dos funciones: `uno()` y `dos()`. Suponga que `main()` llama a `uno()`, y a continuación `uno()` llama a `dos()`. En alguna parte del proceso, `dos()` entonces llama a `uno()` —una segunda llamada a `uno()`—. Esta acción es recursión indirecta, pero es recursiva, ya que `uno()` ha sido llamada dos veces, sin retornar nunca a su llamadora.

Un proceso recursivo debe tener una condición de terminación, ya que si no puede continuar indefinidamente.

Un algoritmo típico que conduce a una implementación recursiva es el cálculo del factorial de un número. El factorial de *n* (`n!`).

```
n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1
```

En consecuencia, el factorial de `4` es igual a `4*3*2*1`, el factorial de `3` es igual a `3*2*1`. Así pues, el factorial de `4` es igual a cuatro veces el factorial de `3`. La Figura 6.10 muestra la secuencia de sucesivas invocaciones a la función factorial.

Ejemplo 6.9

Realizar el algoritmo de la función factorial.

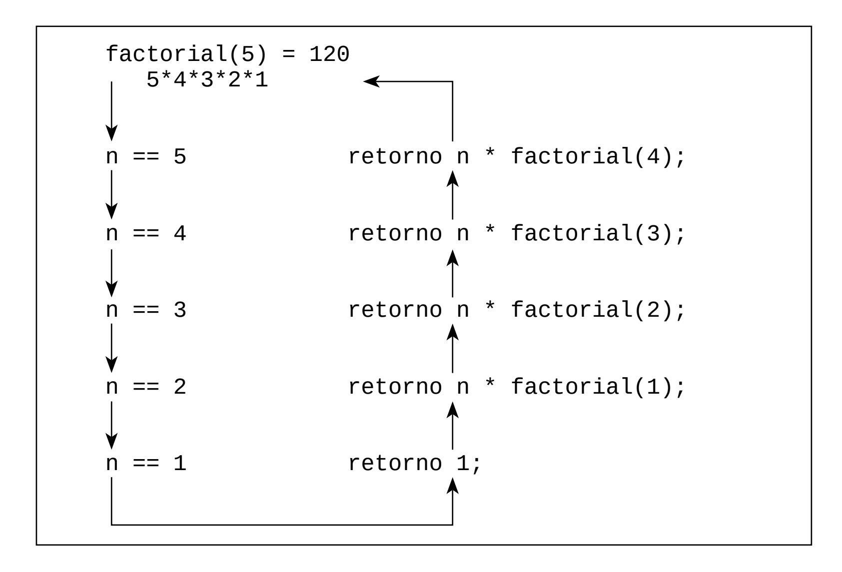


Figura 6.10. Llamadas a funciones recursivas para `factorial(5)`.

La implementación de la función recursiva `Factorial` es:

```
double Factorial(int numero)
{
   if (numero > 1)
      return numero * Factorial(numero-1);
   return 1;
}
```

Ejemplo 6.10

Contar valores de 1 a 10 de modo recursivo.

```
#include <iostream.h>

void Contar(int cima);

int main()
{
   Contar(10);
   return 0;
}
void Contar(int cima)
{
   if (cima > 1)
      Contar(cima-1);
   cout << cima;
}
```

6.19. PLANTILLAS DE FUNCIONES

Las **plantillas de funciones** (*function templates*) proporcionan un mecanismo para crear una *función genérica*. Una función genérica es una función que puede soportar simultáneamente diferentes tipos de datos para su parámetro o parámetros.

Las plantillas son una de las grandes aportaciones de C++. La mayoría de los fabricantes de compiladores (Borland, Microsoft...) o de software libre (DevC++, ...) incorporan *plantillas (templates)*.

Las plantillas son muy útiles para los programadores y se emplean cuando se necesita utilizar la misma función, pero con diferentes tipos de argumentos. Los programadores que utilizan el lenguaje C resuelven este problema normalmente con macros; sin embargo, esta solución no es eficiente.

Consideremos, por ejemplo, una función `abs()` que devuelva el valor absoluto de su parámetro. (El valor absoluto de un número x es x si x es mayor que 0, o $-x$ si x es menor que 0.) Para implementar `abs()` de modo que puedan aceptar parámetros `int`, `float`, `long` y `double`, se han de escribir cuatro funciones distintas; sin embargo, como C++ soporta funciones sobrecargadas, es posible entonces sobrecargar la función `abs` para admitir los cuatro tipos de datos citados.

```
int abs(int x)
{
   return (x < 0) ? -x : x;
}
```

```
long abs(long x)
{
   return (x < 0) ? -x : x;
}

float abs(float x)
{
   return (x < 0) ? -x : x;
}
double abs(double x)
{
   return (x < 0) ? -x : x;
}
```

Para este tipo de problemas, sin embargo, C++ proporciona un mecanismo más ágil y potente, las *plantillas de funciones*. En lugar de escribir las cuatro funciones `abs()`, se puede escribir una única plantilla de función, que se diseña como

```
template <class tipo>
tipo abs(tipo x)
{
   return (x < 0) ? -x : x;
};
```

La palabra reservada `template` indica al compilador que una plantilla de función se ha definido. El símbolo *tipo* indica al compilador que puede ser sustituido por el tipo de dato apropiado: `int`, `float`, etc.

Una plantilla de función tiene el siguiente formato:

```
template <class tipo>
   declaración de la función
```

o expresado más en detalle

```
template <class tipo> tipo función (tipo arg1, tipo arg2,...)
{
   // Cuerpo de la función
}
```

Algunas declaraciones típicas son:

```
1. template <class T> T f(int a, T b)
         {
      // cuerpo de la función
   }

2. template <class T> T max (T a, T b)
   {

      return a > b ? a : b;
   }
```

La plantilla función `max` se declara con un único tipo genérico *T* y con dos argumentos.

3. `template <class T1, class T2> T1 max (T1 a, T2 b)`

La plantilla función `max` se declara con dos tipos diferentes posibles. Devuelve un valor de tipo *`T1`* y requiere dos argumentos: uno de tipo *`T1`* y otro de tipo *`T2`*.

6.19.1. Utilización de las plantillas de funciones

La primera operación a realizar consistirá en diseñar una plantilla de función con uno o varios tipos genéricos *T*. Una vez diseñada la plantilla de funciones, que puede servir para manejar cualquier tipo de datos, tales como datos simples, estructuras, punteros, etc., se puede utilizar ya la plantilla de funciones con un tipo de dato específico, con lo que se crea una *función de plantillas* (éstos son términos confusos, pero son los mismos que Stroustrup utiliza en su libro de C++).

Así, por ejemplo, consideremos la función `Func1()`

```
template <class Tipo> Tipo Func1(Tipo arg1, Tipo arg2)
{
   // cuerpo de la función Func1()
}

void main()
{
   int i, j;      // Declarar variables enteros

   float a, b;    // Declarar variables float

   // uso de la función Func1() con valores enteros
   Func1(i, j);

   // uso de la función Func2() con variables float
   Func1(a, b);
}
```

El programa anterior declara una función plantilla `Func1()` y utiliza la función con argumentos enteros (`int`) o reales (`float`).

Cuando el compilador ve la función plantilla, no sucede nada hasta que la función se utiliza realmente en el programa. Cuando la función `Func1()` se utiliza por primera vez, se llama con dos argumentos enteros. El compilador examina la función plantilla, observa que los argumentos se declaran en variables de tipos genéricos y se construye una función correspondiente que utiliza enteros como argumentos. El compilador genera la siguiente función real de la función plantilla:

```
int Func1(int arg1, int arg2){
   // cuerpo de la función Func1()
}
```

El tipo de objeto `int` se utiliza en cualquier parte que la variable `Tipo` estaba declarada. Cuando la función `Func1()` se utiliza la segunda vez con argumentos float, el compilador genera la siguiente función real de la función plantilla:

```
float Func1(float arg1, float arg2)
{
   // cuerpo de la función Func1()
}
```

6.19.2. Plantillas de función `min` y `max`

El listado `minmax.h` declara dos plantillas de funciones: `min()` y `max()`.

```
//minmax.h

#ifndef _ _MINMAX_H
#define _ _MINMAX_H      // Evita inclusiones múltiples

template <class T> T max(T a, T b)
{
   if (a > b)
      return a;
   else
      return b;
}

template <class T> T min(T a, T b)
{
   if (a < b)
      return a;
   else
      return b;
}

#endif
```

Un programa que utiliza las funciones de plantilla, mediante prototipos que el compilador utiliza para escribir los cuerpos de las funciones reales:

```
// Archivo PLANTMMX.CPP

#include <iostream>
#include "minmax.h"
using namespace std;

int max(int a, int b);
double max(double a, double b);
char max(char a, char b);

main()
{
   int e1 = 400; e2 = 500;
   double d1 = 2.718283, d2 = 3.141592;
   char c1 = 'X', c2 = 't';

   cout << "maximo(e1, e2) = " << max(e1, e2) << endl;
   cout << "maximo(d1, d2) = " << max(d1, d2) << endl;
   cout << "maximo(c1, c2) = " << max(c1, c2) << endl;
   return 0;
}
```

RESUMEN

Las funciones son la base de la construcción de programas en C++. Se utilizan funciones para subdividir problemas grandes en tareas más pequeñas. El encapsulamiento de las características en funciones, hace los programas más fáciles de mantener. El uso de funciones ayuda al programador a reducir el tamaño de su programa, ya que se puede llamar repetidamente y reutilizar el código dentro de una función. En este capítulo habrá aprendido lo siguiente:

- el concepto, declaración, definición y uso de una función;
- las funciones que devuelven un resultado lo hacen a través de la sentencia `return`;
- los parámetros de funciones se pueden pasar por referencia y por valor;
- el modificador `const` se utiliza cuando se desea que los parámetros de la función sean valores de sólo lectura;
- el concepto y uso de prototipos, cuyo uso es obligatorio en C++;
- el concepto de argumentos por omisión y cómo utilizarlos en las funciones;
- la ventaja de utilizar funciones en línea (**`inline`**) para aumentar la velocidad de ejecución;
- el concepto de *ámbito* o *alcance* y *visibilidad,* junto con el de *variable global* y *local*;
- clases de almacenamiento de variables en memoria: `auto`, `extern`, `register` y `static`.

La biblioteca estándar C++ de funciones en tiempo de ejecución incluye gran cantidad de funciones. Se agrupan por categorías, entre las que destacan:

- manipulación de caracteres;
- numéricas;
- tiempo y hora;
- conversión de datos;
- búsqueda y ordenación;
- etcétera.

Tenga cuidado de incluir el archivo de cabecera correspondiente cuando desee incluir funciones de biblioteca en sus programas.

Una de las características más sobresalientes de C++ que aumentan considerablemente la potencia de los programas es la posibilidad de manejar las funciones de modo eficiente, apoyándose en la propiedad que les permite ser compiladas por separado.

Otros temas tratados han sido:

- *Ámbito* o las *reglas de visibilidad* de funciones y variables.
- El entorno de un programa tiene cuatro tipos de ámbito: de programa, archivo fuente, función y bloque. Una variable está asociada a uno de esos ámbitos y es invisible (no accesible) desde otros ámbitos.
- Las *variables globales* se declaran fuera de cualquier función y son visibles a todas las funciones. Las variables globales se declaran dentro de una función y sólo pueden ser utilizadas por esa función.

```
int i;          // variable global,
                   ámbito de programa
static int j  // ámbito de archivo

void main()
{
  int d, e;    // variable local, ám-
                   bito de función
  // ...
}

func(int j)
{
  if (j > 3)
  {
    int i;    // ámbito de bloque
    for (i = 0; i < 20; i++)
      func2(i);
  }
  // i no es visible ya
}
```

- *Variables automáticas* son las variables, por defecto, declaradas localmente en una función.
- *Variables estáticas* mantienen su información, incluso después de que la función ha terminado. Cuando se llama de nuevo la función, la variable se pone al valor que tenía cuando se llamó anteriormente.
- *Funciones recursivas* son aquellas que se pueden llamar a sí mismas.
- C++ permite *sobrecarga de funciones* o *polimorfismo de funciones*, que permite utilizar múltiples funciones con el mismo nombre pero diferente lista de argumentos.
- Por último, se han considerado *plantillas de funciones* que proporcionan el mecanismo para crear funciones genéricas. Una función genérica es una función que puede soportar simultáneamente diferentes tipos de datos en sus parámetros.
- Las *variables registro* se pueden utilizar cuando se desea aumentar la velocidad de procesamiento de ciertas variables.

EJERCICIOS

6.1. Escribir una función lógica *Dígito* que determine si un carácter es uno de los dígitos de 0 a 9.

6.2. Escribir una función Redondeo que acepte un valor real Cantidad y un valor entero Decimales y devuelva el valor Cantidad redondeado al número especificado de Decimales. Por ejemplo, Redondeo (20.563,2) devuelve los valores 20.0, 20.5 y 20.54 respectivamente.

6.3. Escribir un programa que permita al usuario elegir el cálculo del área de cualquiera de las figuras geométricas: círculo, cuadrado, rectángulo o triángulo, mediante funciones.

6.4. Determinar y visualizar el número más grande de dos números dados, mediante una función

6.5. Escribir una función recursiva que calcule los *N* primeros números naturales.

PROBLEMAS

6.1. Escribir un programa que solicite del usuario un carácter y que sitúe ese carácter en el centro de la pantalla. El usuario debe poder a continuación desplazar el carácter pulsando las letras A (arriba), B (abajo), I (izquierda), D (derecha) y F (fin) para terminar.

6.2. Escribir una función que reciba una cadena de caracteres y la devuelva (en parámetros) en forma inversa (`'hola'` se convierte en `'aloh'`).

6.3. Escribir una función que determine si una cadena de caracteres es un palíndromo (un palíndromo es un texto que se lee igual en sentido directo y en inverso: radar).

6.4. Escribir un programa mediante una función que acepte un número de día, mes y año y lo visualice en el formato.

```
dd//mm/aa
```

Por ejemplo, los valores 8, 10 y 1946 se visualizan como

```
8/10/46
```

6.5. Escribir un programa que lea un entero positivo y, a continuación, llame a una función que visualice sus factores primos.

6.6. Escribir un programa, mediante funciones, que visualice un calendario de la forma:

```
L    M    M    J    V    S    D
          1    2    3    4    5
6    7    8    9    10   11   12
13   14   15   16   17   18   19
20   21   22   23   24   25   26
27   28   29   30
```

El usuario indica únicamente el mes y el año. La fórmula que permite conocer el día de la semana correspondiente a una fecha dada es:
meses de enero o febrero

```
n = a+31*(m-1)+d(a-1)div 4-3*((a+99)
    div 100) div 4;
```

restantes meses

```
n = a+31*(m-1)+d-(4*m+23)/div 10+a
    div 4-(3*(a div/100+1))div/4;
```

donde *a* = año, *m* = mes, *d* = día.

Nota: / indica división entera

n `% 7` indica el día de la semana (1 = lunes, 2 = martes, etc.).

6.7. Escribir un programa que lea los dos enteros positivos *n* y *b* que llame a una función `CambiarBase` para calcular y visualizar la representación del número *n* en la base *b*.

6.8. Escribir un programa que permita el cálculo del *mcd* (máximo común divisor) de dos números por el algoritmo de Euclides. (Dividir a entre b, se obtiene el cociente q y el resto r si es cero b es el mcd, si no se divide b entre r, y así sucesivamente hasta encontrar un resto cero, el último divisor es el *mcd*.)

6.9. Escribir el inverso de un número entero dado (1234, inverso 4321).

6.10. Escribir una función que permita calcular la serie:

$$1^2 + 2^2 + 3^2 + ... + n^2 = \frac{n.(n + 10).\ (2n + 1)}{6}\ n =$$

$$= * (n + 10) * (2 * n{+}1) / 6$$

6.11. Escribir un programa que lea dos números x y n y calcule la suma de la progresión geométrica.

$$1 + x + x^2 + x^3 + ... + \ x^n$$

6.12. Escribir un programa que encuentre el valor mayor, el valor menor y la suma de los datos de entrada. Obtener la media de los datos mediante una función.

6.13. Escribir un programa que utilice una función Potencia para calcular el pago mensual de un préstamo dado por la fórmula:

$$\frac{r.A/n}{1 - (1r/n^{-n.y}}$$

6.14. Escribir una función que acepte un parámetro x(x≠0) y devuelva el siguiente valor:

$$\frac{1}{x^5\left(\frac{e^{1.435}}{x} - 1\right)}$$

6.15. Escribir una función con dos parámetros, x y n, que devuelva lo siguiente:

$$x + \frac{x^n}{n} - \frac{x^{n+2}}{n+2} \qquad \text{si } x > = 0$$

$$\frac{x^{n+1}}{n+1} - \frac{x^{n-1}}{n-1} \qquad \text{si } x < = 0$$

6.16. Escribir una función que tome como parámetros las longitudes de los tres lados de un triángulo (a, b y c) y devuelva el área del triángulo.

$$\text{Área} = \sqrt{p(p-a)(p-b)(p-c)} \quad \text{donde} \quad p = \frac{a+b+c}{2}$$

6.17. Escribir un programa mediante funciones que realicen las siguientes tareas:

a) Devolver el valor del día de la semana en respuesta a la entrada de la letra inicial (mayúscula o minúscula) de dicho día.
b) Determinar el número de días de un mes y año dados.

6.18. Escribir un programa que lea una cadena de hasta diez caracteres que representa a un número en numeración romana e imprima el formato del número romano y su equivalente en numeración arábiga. Los caracteres romanos y sus equivalentes son:

M	1000	L	50
D	500	X	10
C	100	V	5
		I	1

Compruebe su programa para los siguientes datos:

LXXXXVI (86), CCCXIX (319), MCCLIV (1254)

6.19. Escriba una función que calcule cuántos puntos de coordenadas enteras existen dentro de un triángulo del que se conocen las coordenadas de sus tres vértices.

6.20. Codifique una función que escriba N líneas en blanco.

6.21. Escribir un programa que mediante funciones determine el área del círculo correspondiente a la circunferencia circunscrita de un triángulo del que conocemos las coordenadas de los vértices.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 126).
2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

6.1. *Escribir una función que tenga un argumento de tipo entero y que devuelva la letra P si el número es positivo, y la letra N si es cero o negativo.*

6.2. *Escribir una función lógica de dos argumentos enteros, que devuelva* `true` *si uno divide al otro y* `false` *en caso contrario.*

6.3. *Escribir una función que convierta una temperatura dada en grados Celsius a grados Fahrenheit. La fórmula de conversión es:*

$$F = \frac{9}{5} C + 32$$

6.4. *Escribir una función lógica* `Vocal` *que determine si un carácter es una vocal.*

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 127).
2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

6.1. *Escribir una función que tenga como parámetro dos números enteros positivos* `num1` *y* `num2`*, y calcule el resto de la división entera del mayor de ellos entre el menor mediante suma y restas.*

6.2. *Escribir una función que tenga como parámetros dos números enteros positivos* `num1` *y* `num2` *y calcule el cociente de la división entera del primero entre el segundo mediante sumas y restas.*

6.3. *Escribir un programa que calcule los valores de la función definida de la siguiente forma:*

```
funciony(0) = 0,
funciony(1) = 1
funciony(2) = 2
funciony(n) = funciony(n – 3) + 3
              *funciony(n – – 2) –
              funcionx(n – 1) si
              n > 2.
```

6.4. *Un número entero* `n` *se dice que es perfecto si la suma de sus divisores incluyendo 1 y excluyéndose él coincide consigo mismo. Codificar una función que decida si un número es perfecto. Por ejemplo 6 es un número perfecto* `1 + 2 + 3 = 6`.

6.5. *Escribir una función que decida si dos números enteros positivos son amigos. Dos números son amigos, si la suma de los divisores distintos de sí mismo de cada uno de ellos coincide con el otro número. Ejemplo* `284` *y* `220` *son dos números amigos.*

6.6. *Dado el valor de un ángulo, escribir una función que muestra el valor de todas las funciones trigonométricas correspondientes al mismo.*

6.7. *Escribir una función que decida si un número entero positivo es primo.*

6.8. *Escribir una función para calcular las coordenadas* `x` *e* `y` *de la trayectoria de un proyectil de acuerdo a los parámetros ángulo de inclinación* `alfa` *y velocidad inicial* `v` *a intervalos de* `0.1` *s.*

6.9. *Escribir un programa que mediante funciones calcule:*

- *Las anualidades de capitalización si se conoce el tiempo, el tanto por ciento y el capital final a pagar.*
- *El capital c que resta por pagar al cabo de t años conociendo la anualidad de capitalización y el tanto por ciento.*
- *El número de años que se necesita para pagar un capital c a un tanto por ciento r.*

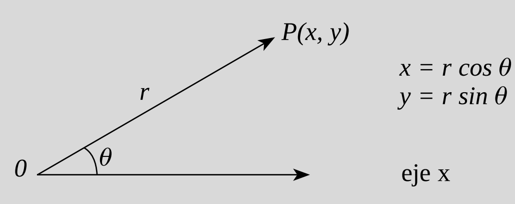
6.10. *Se define el número combinatorio m sobre n de la siguiente forma:* $\binom{m}{n} = \frac{m!}{n!(m-n)!}$. *Escribir un programa que lea los valores de m y de n y calcule el valor de m sobre n.*

6.11. *Dado un número real* `p` *entre cero y uno, un número entero* `n` *positivo, y otro número entero* `i` *comprendido entre* `0` *y* `n`, *se sabe que si un suceso tiene probabilidad de que ocurra* `p`, *y el experimento aleatorio se repite* `n` *veces, la probabilidad de que el suceso ocurra* `i` *veces viene dado por la función* `binomial` *de parámetros n, p e i dada por la siguiente fórmula:*

$$Probabilidad(X = i) = \binom{n}{i} p^i (1 - p)^{n-i}$$

Escribir un programa que lea los valores de `p`, `n` *e* `i`, *y calcule el valor dado por la función* `binomial`.

6.12. *Escribir un programa que utilice una función para convertir coordenadas polares a rectangulares.*



6.13. *La ley de probabilidad de que ocurra el suceso r veces de la distribución de Poisson de media m viene dado por:*

$$Probabilidad(X = r) = \frac{m^r}{r!} e^{-m}$$

Escribir un programa que calcule mediante un menú el valor de:

a) *El suceso ocurra exactamente r veces.*
b) *El suceso ocurra a lo sumo r veces.*
c) *El suceso ocurra por lo menos r veces.*

6.14. *Escribir funciones que calculen el máximo común divisor y el mínimo común múltiplo de dos números enteros.*

6.15. *Escribir funciones para leer fracciones y visualizarlas, representando las fracciones por dos números enteros numerador y denominador.*

6.16. *Escribir un programa que mediante un menú permita gestionar las operaciones suma resta, producto y cociente de fracciones.*

6.17. *La descomposición en base 2 de todo número, permite en particular que todo número en el intervalo [1,2], se pueda escribir como límite de la serie* $\sum_{i=1}^{n} \Downarrow sg(i) \frac{1}{2^i}$ *donde la elección del signo sg(i) depende del número que se trate.*

El signo del primer término es siempre positivo. Una vez calculado los signos de los n primeros, para calcular el signo del siguiente término se emplea el esquema: signo es positivo $sg(n + 1) = +1$ *si se cumple:*

$$2^{\sum_{i=1}^{n} sg(i) \frac{1}{2^i}} > x$$

en caso contrario, $sg(n + 1) = -1$.

Escribir un programa que calcule el logaritmo en base dos de un número $x > 0$ *con un error absoluto menor o igual que épsilon (x y épsilon son datos).*

6.18. *La función seno (sen) viene definida mediante el siguiente desarrollo en serie.*

$$sen(x) = \sum_{i=0}^{n} \frac{x^{2i+1}}{(2i+1)!}$$

Escribir una función que reciba como parámetro el valor de x así como una cota de error, y calcule el seno de x con un error menor que la cota que se le pase. Ejecute la función en un programa con varios valores de prueba.

6.19. *La función coseno viene definida mediante el siguiente desarrollo en serie de Taylor.*

$$cos(x) = \sum_{i=0}^{n} \frac{x^{2i}}{(2i)!}$$

Escribir una función que reciba como parámetro el valor de x así como una cota de error, y calcule el coseno de x con un error menor que la cota que se le pase.

6.20. *La función* `clotoide` *viene definida por el siguiente desarrollo en serie, donde* `A` *y* K *son datos.*

$$x = A\sqrt{2K} \sum_{i=0}^{n} (-1)^i \frac{K^{2i}}{(4i + 1)(2i)!}$$

$$y = A\sqrt{2K} \sum_{i=0}^{n} (-1)^i \frac{K^{2i+1}}{(4i + 3)(2i + 1)!}$$

Escribir un programa que calcule los valores de la clotoide para el valor de $A = 1$ *y para los valores de* θ *siguientes* $0, \pi/20, 2\pi/20, 3\pi/20, ..., \pi$*. La parada de la suma de la serie, será cuando el valor absoluto del siguiente término a sumar sea menor o igual que* $1\ e^{-10}$.

CAPÍTULO 7

Arrays/arreglos (listas y tablas)

Contenido

INTRODUCCIÓN

En capítulos anteriores se han descrito las características de los tipos de datos básicos o simples (carácter, entero y coma flotante). Asimismo, se ha aprendido a definir y utilizar constantes simbólicas utilizando `const`, `#define` y el tipo `enum`. En este capítulo continuaremos el examen de los restantes tipos de datos de C++, examinando especialmente el tipo *array* o **arreglo** (lista o tabla), la estructura, la unión y una breve introducción al objeto.

En este capítulo aprenderá el concepto y tratamiento de los arrays. Un array almacena muchos elementos del mismo tipo, tales como veinte enteros, cincuenta números de coma flotante o quince caracteres. El *array* es muy importante por diversas razones, una de ellas es almacenar secuencias o *cadenas* de texto. Hasta el momento C++ proporciona datos de un solo carácter; utilizando el tipo *array*, se puede crear una variable que contenga un grupo de caracteres.

CONCEPTOS CLAVE

- Array.
- Arrays bidimensionales.
- Array de caracteres.
- Arrays multidimensionales.
- Cadena de texto.
- Declaración de un array.
- Iniciación de un array.
- Lista, tabla.
- Ordenación y búsqueda.
- Parámetros de tipo array.

7.1. ARRAYS (ARREGLOS)

Un *array* o **arreglo** (lista o tabla) es una secuencia de objetos del mismo tipo. Los objetos se llaman elementos del *array* y se numeran consecutivamente 0, 1, 2, 3... El tipo de elementos almacenados en el *array* puede ser cualquier tipo de dato de C++, incluyendo estructuras definidas por el usuario, como se describirá más tarde. Normalmente, el *array* se utiliza para almacenar tipos tales como `char`, `int` o `float`.

Un *array* puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada, o el número de personas que residen en cada una de las diecisiete comunidades autónomas españolas. Cada item del *array* se denomina *elemento*.

Los elementos de un *array* se numeran, como ya se ha comentado, consecutivamente 0, 1, 2, 3... Estos números se denominan *valores índice* o *subíndice* del *array*. El término *subíndice* se utiliza ya que se especifica igual que en matemáticas como una secuencia tal como a_0, a_1, a_2... Estos números localizan la posición del elemento dentro del *array*, proporcionando *acceso directo* al *array*.

Si el nombre del *array* es `a`, entonces `a[0]` es el nombre del elemento que está en la posición 0, `a[1]` es el nombre del elemento que está en la posición 1, etc. En general, el elemento *i-ésimo* está en la posición *i-1*. De modo que si el *array* tiene *n* elementos, sus nombres son `a[0], a[1],...,a[n-1].` Gráficamente se representa así el *array* `a` con seis elementos (Figura 7.1).

El *array* `a` tiene 6 elementos: `a[0]` contiene 25,1, `a[1]` contiene 34,2, `a[2]` contiene 5,25, `a[3]` contiene 7,45, `a[4]` contiene 6,09 y `a[5]` contiene 7,54. El diagrama de la Figura 7.1 representa realmente una región de la memoria de la computadora ya que un *array* se almacena siempre con sus elementos en una secuencia de posiciones de memoria contigua.

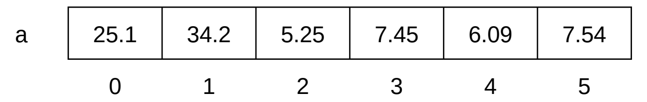


Figura 7.1. *Array* de seis elementos.

7.1.1. Declaración de un array

Al igual que con cualquier tipo de variable, se debe declarar un array antes de utilizarlo. Un array se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador el *tamaño* o *longitud* del array. Para indicar al compilador el *tamaño* o *longitud* del array se debe hacer seguir al nombre, el tamaño encerrado entre corchetes. La *sintaxis* para declarar un array de una dimensión determinada es:

```
tipo nombreArray[numeroDeElementos];
```

númeroDeElementos tamaño del array; debe ser; una constante entera, tal como 10; un valor `const`, o una expresión constante, tal como `5 * sizeof(int)`; no puede ser una variable cuyo valor se establezca mientras se ejecuta el programa.

Por ejemplo, para crear un array (lista) de diez variables enteras, se escribe:

```
int numeros[10]; //Crea un array de 10 elementos int
```

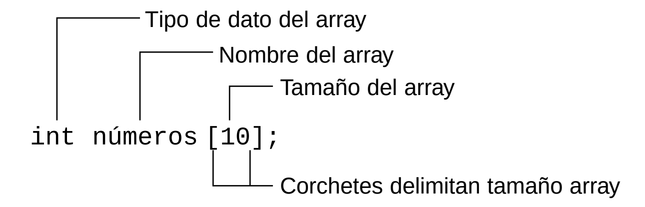


Figura 7.2. Sintaxis de la definición de un array.

Esta declaración hace que el compilador reserve espacio suficiente para contener diez valores enteros. En C++ los enteros ocupan, normalmente, 2 bytes, de modo que un array de diez enteros ocupa 20 bytes de memoria. La Figura 7.3 muestra el esquema de un array de diez elementos; cada elemento puede tener su propio valor.

Array de datos enteros: a

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Un array de enteros se almacena en bytes consecutivos de memoria. Cada elemento utiliza dos bytes. Se accede a cada elemento de array mediante un índice que comienza en cero. Así, el elemento quinto (`a[4]`) del array ocupa los bytes 9.° y 10.°.

Figura 7.3. Almacenamiento de un array en memoria.

7.1.2. Acceso a los elementos de un array

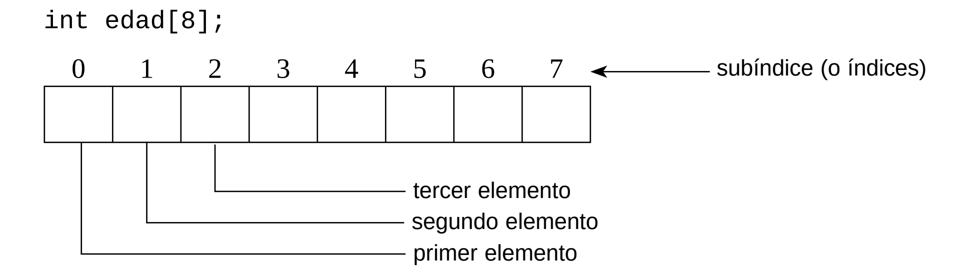
Gran parte de la utilidad de un array proviene del hecho que se pueda acceder a los elementos de dicho array de modo individual. El método para acceder a un elemento es utilizar un *subíndice* o un *índice*, con la sintaxis siguiente:

nombreArray[n]

número del elemento dentro del array

El primer elemento del array tiene de índice 0, el siguiente 1, y así sucesivamente.

```
int edad[8];
```



El array (lista o vector) `edad` tiene 8 valores, cada uno de los cuales es una variable de un tipo entero.

Los arrays siempre comienzan en el elemento 0. Así pues, el array `números` contiene los siguientes elementos individuales:

```
numeros[0]   numeros[1]   numeros[2]   numeros[3]
numeros[4]   numeros[5]   numeros[6]   numeros[7]
numeros[8]   numeros[9]
```

Por ejemplo,

```
cout << numeros[4] << endl;
```

visualiza el valor del elemento 5 del array `numeros`.

Precaución

C++ no comprueba que los índices del array están dentro del rango definido. Así, por ejemplo, se puede intentar acceder a `numeros[12]` y el compilador no producirá ningún error, lo que puede producir un fallo en su programa, dependiendo del contexto en que se encuentre el error.

El término *índice* o *subíndice* procede de las matemáticas, en las que un subíndice se utiliza para representar un elemento determinado.

$numeros_0$ *equivale a* `numeros[0]`
$numeros_3$ *equivale a* `numeros[3]`

El método de numeración del elemento *i-ésimo* con el índice o subíndice *i-1* se denomina *indexación basada en cero*. Su uso tiene el efecto de que el índice de un elemento del array es siempre el mismo que el número de «pasos» desde el elemento inicial `a[0]` a ese elemento. Por ejemplo, `a[3]` está a 3 pasos o posiciones del elemento `a[0]`. La ventaja de este método se verá de modo más evidente al tratar las relaciones entre arrays y punteros (Capítulo 9).

Ejemplos

`int edad[5];`	*Array* `edad` *contiene 5 elementos: el primero,* `edad[0]` *y el último,* `edad[4]`.
`int pesos[25], longitudes[100];`	*Declara 2 arrays de enteros.*
`float salarios[25];`	*Declara un array de 25 elementos* `float`.
`double temperaturas[50];`	*Declara un array de 50 elementos* `double`.
`char letras[15];`	*Declara un array de caracteres.*

En los programas se pueden referenciar elementos utilizando fórmulas para los subíndices. Mientras que el subíndice puede evaluar a un entero, se puede utilizar una constante, una variable o una expresión para el subíndice. Así, algunas referencias individuales a elementos son:

```
edad[4]
ventas[total + 5]
bonos[mes]
salario[mes[i] * 5]
```

Ejemplo 7.1

```
// demoArreglo.cpp
// obtener 10 edades de alumnos y visualizarlas
#include <iostream>
using namespace std;

int main()
{
```

```
    int edad[10];   //arreglo de 10 enteros
    for (int i = 0; i < 10; i++)
    {
       cout << "Introduzca edad del alumno: ";
       cin >> edad[i];
    }

    for (i = 0; i < 10; i++)
       cont << "las edades son " << edad[i] << endl;
    return 0;
}
```

7.1.3. Almacenamiento en memoria de los arrays

Los elementos de los arrays se almacenan en bloques contiguos. Así, por ejemplo, los arrays

```
int edades[5];
char codigos[5];
```

se representan gráficamente en memoria en la Figura 7.4.

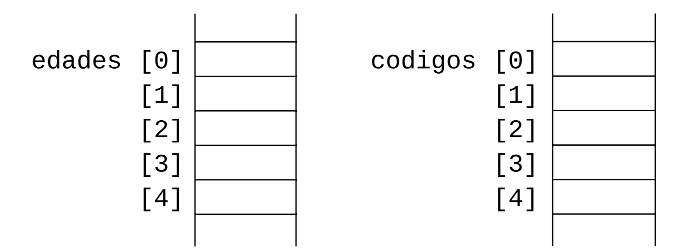


Figura 7.4. Almacenamiento en memoria de arrays.

Nota

Todos los subíndices de los arrays comienzan con 0.

Precaución

C++ permite asignar valores fuera de rango a los subíndices. Se debe tener cuidado no hacer esta acción, debido a que se sobreescribirían datos o código.

Los arrays de caracteres funcionan de igual forma que los arrays numéricos, partiendo de la base de que cada carácter ocupa normalmente un byte. Así, por ejemplo, un array llamado `nombre` se puede representar en la Figura 7.5.

```
char nombre[] = "Mortimer"
char nombre[10] = "Mackoy"
```

nombre	
M	[0]
o	[1]
r	[2]
t	[3]
i	[4]
m	[5]
e	[6]
r	[7]

Figura 7.5. Almacenamiento de un array de caracteres en memoria.

7.1.4. El tamaño de los arrays (`sizeof`)

La función `sizeof()` devuelve el número de bytes necesarios para contener su argumento. Si se usa `sizeof()` para solicitar el tamaño de un array, esta función devuelve el número de bytes reservados para el array completo.

Por ejemplo, supongamos que se declara un array de enteros de 100 elementos denominado `edades`; si se desea producir el tamaño del array, se puede utilizar una sentencia similar a:

```
n = sizeof(edades);
```

donde *n* tomará el valor 200. Si se desea solicitar el tamaño de un elemento individual del array, tal como

```
n = sizeof(edades[6]);
```

n almacenará el valor 2 (número de bytes que contienen un entero).

7.1.5. Verificación del rango del índice de un array

C++ al contrario que otros lenguajes de programación —por ejemplo, Pascal— no verifica el valor del índice de la variable que representa al array. Así, por ejemplo, en Pascal si se define un array `m` con índices 0 a 5, entonces `a[6]` hará que el programa se «rompa» («produzca un error»).

Ejemplo 7.2

Protección frente a errores en el intervalo (rango) de valores de una variable de índice que representa un array.

```
double suma(const double a[], const int n)
{
   if (n * sizeof(double) > sizeof(a)
      return 0;
   double S = 0.0;
   for (int i = 0; i < n; i++)
      S += a[i];
   return S;
}
```

7.2. *INICIALIZACIÓN* (INICIACIÓN) DE UN ARRAY (ARREGLO)

Se deben asignar valores a los elementos del array antes de utilizarlos, tal como se asignan valores a variables. Para asignar valores a cada elemento del array de enteros `precios`, se puede escribir:

```
precios[0] = 10;
precios[1] = 20;
precios[3] = 30;
precios[4] = 40;
...
```

La primera sentencia fija `precios[0]` al valor 10, `precios[1]` al valor 20, etc. Sin embargo, este método no es práctico cuando el array contiene muchos elementos. El método utilizado normalmente es *inicializar* (iniciar) el array completo en una sola sentencia.

Se pueden dar valores a los elementos de un array cuando se define. El sistema consiste en almacenar los valores correspondientes encerrados entre llaves y separados por comas (Figura 7.6).

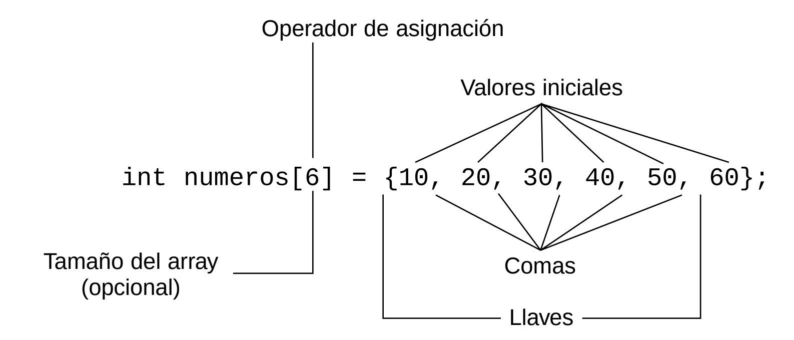


Figura 7.6. Sintaxis de inicialización de un array o arreglo.

Realmente, no se necesita utilizar el tamaño del array cuando se inicializan los elementos del array, ya que el compilador contará el número de variables que inicializa.

Cuando se inicializa un array, el tamaño del array se puede determinar automáticamente por las constantes de inicialización. Estas constantes se separan por comas y se encierran entre llaves, como en los siguientes ejemplos:

```
int numeros[6] = {10, 20, 30, 40, 50, 60};
int n[] = {3, 4, 5}           //Declara un array de 3 elementos
char c[] = {'L','u','i','s'}; //Declara un array de 4 elementos
```

El array `numeros` tiene 6 elementos, `n` tiene 3 elementos y el array `c` tiene 4 elementos. Los arrays de caracteres se pueden inicializar con una constante de cadena, como en

```
char s[] = {"Mortimer"};
```

Nota

C++ puede dejar los corchetes vacíos, sólo cuando se asignan valores al array, tal como

```
int cuenta[] = {15, 25, -45, 0, 50};
```

El compilador asigna automáticamente cinco elementos a cuenta.

El método de inicializar arrays mediante valores constantes después de su definición es adecuado cuando el número de elementos del array es pequeño. Por ejemplo, para inicializar un array (lista) de 10 enteros a los valores `10` a `1` y, a continuación, visualizar dichos valores en un orden inverso, se puede escribir:

```
int cuenta[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1,};
for (i = 9; i >= 1; i—)
   cout << "\n cuenta descendente" << i << "=" << cuenta[i];
```

Se pueden asignar constantes simbólicas como valores numéricos, de modo que las sentencias siguientes son válidas:

```
const int ENE = 31, FEB = 28, MAR = 31, ABR = 30, MAY = 31,
   JUN = 30, JUL = 31, AGO = 31, SEP = 30, OCT = 31,
   NOV = 30, DIC = 31;
...
int meses[12] =    {ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO,
                    SEP, OCT, NOV,DIC};
```

Se pueden asignar valores a un array utilizando un bucle `for` o `while/do-while`, y éste suele ser el sistema más empleado normalmente. Por ejemplo, para inicializar todos los valores del array `números` al valor 0, se puede utilizar la siguiente sentencia:

```
for (i = 0; i <= 5; i++)
   numeros[i] = 0;
```

debido a que el valor del subíndice `i` varía de `0` a `5`, cada elemento del array `numeros` se inicializa y establece a cero.

Ejemplo 7.3

El programa `INICIALI.CPP` asigna ocho enteros a un array `nums`, mediante `cin`; a continuación, visualiza el total de los números.

```
#include <iostream>
using namespace std;

const NUM 8

main()
{
   int nums[NUM];
   int total = 0;

   for (int i = 0; i < NUM; i++)
   {
      cout << "Por favor, introduzca el número";
      cin >> nums[i];
      total += nums[i];
   }

   cout << "El total de números es" << total << endl;
   return 0;
}
```

Las variables globales que representan arrays se inicializan a `0` por defecto. Por ello, la inicialización siguiente no es correcta, y se visualiza `0` para los 10 valores del array al ejecutarse el programa.

```
int lista[10];
main()
{
   int j;
   for (j = 0; j <= 9; j++)
      cout << "\n lista[" << j << "] = " << lista[i];
}
```

Así, por ejemplo, en

```
int Notas[5];
void main()
{
   static char Nombres[5];
}
```

> Si se define un array globalmente o un array estático y no se proporciona ningún valor de inicialización, el compilador inicializará el array con un valor por defecto (cero para arrays de elementos enteros y reales —coma flotante— y carácter nulo para arrays de caracteres).

el array de enteros se ha definido globalmente y el array de caracteres se ha definido como un array local estático de `main()`. Si se ejecuta ese segmento de programa, se obtendrán las siguientes asignaciones a los elementos de los arrays:

```
 Notas                  Nombres
[0]   0               [0]     '\0'
[1]   0               [1]     '\0'
[2]   0               [2]     '\0'
[3]   0               [3]     '\0'
[4]   0               [4]     '\0'
```

7.3. ARRAYS DE CARACTERES Y CADENAS DE TEXTO

Una cadena de texto es un conjunto de caracteres, tales como «`ABCDEFG`». C++ soporta cadenas de texto utilizando un array de caracteres que contenga una secuencia de caracteres:

```
char cadena[] = "ABCDEFG";
```

Es importante comprender la *diferencia* entre un array de caracteres y una cadena de caracteres. Las *cadenas* contienen un carácter nulo al final del array de caracteres.

> Las cadenas se deben almacenar en arrays de caracteres, pero no todos los arrays de caracteres contienen cadenas.

Examine la Figura 7.7, donde se muestra una cadena de caracteres y un array de caracteres.

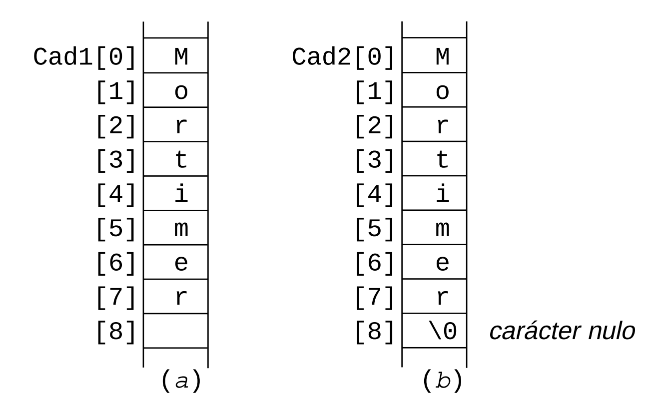


Figura 7.7. (a) Array de caracteres; (b) cadena.

Las cadenas se señalan incluyendo un carácter al final de la cadena: el *carácter nulo* (`\0`), cuyo valor en el código ASCII es `0`. El medio más fácil de inicializar un array de caracteres es hacer la inicialización de la declaración:

```
char Cadena[7] = "ABCDEF";
```

El compilador inserta automáticamente un carácter nulo al final de la cadena, de modo que la secuencia real sería:

```
char Cadena[7] = "ABCDEF"
```

Cadena	A	B	C	D	E	F	\0

La asignación de valores a una cadena se puede hacer del modo siguiente:

```
Cadena[0] = 'A';
Cadena[1] = 'B';
Cadena[2] = 'C';
Cadena[3] = 'D';
Cadena[4] = 'E';
Cadena[5] = 'F';
Cadena[6] = '\0';
```

Sin embargo, no se puede asignar una cadena a un array del siguiente modo:

```
Cadena = "ABCDEF"
```

Para copiar una constante cadena o copiar una variable de cadena a otra variable de cadena se debe utilizar la función de la biblioteca estándar —posteriormente se estudiará— `strcpy()` ("copiar cadenas"). `strcpy()` permite copiar una constante de cadena en una cadena. Para copiar el nombre «`Abracadabra`» en el array `nombre`, se puede escribir

```
strcpy(nombre, "Abracadabra");  // Copia Abracadabra en nombre
```

`strcpy` añade un carácter nulo al final de la cadena. A fin de que no se produzcan errores en la sentencia anterior, se debe asegurar que el array de caracteres nombre tenga elementos suficientes para contener la cadena situada a su derecha.

Ejemplo 7.4

Rellenar los elementos de un array con datos procedentes del teclado.

```
// Rellenado de datos de una lista

#include <iostream>
using namespace std;

// Constantes y variables globales

const int MAX = 10;
int Muestra[MAX];

void main()
{
   cout <<"Introduzca una lista de " << MAX << " elementos y pulse la
           tecla Intro(ENTER) después de cada entrada \n\n";
   for (int i = 0, i < MAX; ++i)
      cin >> Muestra[i];
}
```

Ejemplo 7.5

Visualizar un array después de introducir datos en el mismo.

```
// Visualizar los elementos de un array
#include <iostream>
using namespace std;

// constantes y variables globales
const int MAX = 10;
int Muestra[MAX];

void main()
{
   for (int i = 0; i < MAX; ++i)
      Muestra[i] = i * i;
   for ( i = 0, i < MAX; ++i)
      cout << Muestra[i] << '\t';
}
```

7.4. ARRAYS MULTIDIMENSIONALES

Los arrays vistos anteriormente se conocen como arrays *unidimensionales* (una sola dimensión) y se caracterizan por tener un solo subíndice. Estos arrays se conocen también por el término *listas*. Los *arrays multidimensionales* son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los arrays más usuales son los de dos dimensiones, conocidos también por el nombre de *tablas* o *matrices*. Sin embargo, es posible crear arrays de tantas dimensiones como requieran sus aplicaciones, esto es, tres, cuatro o más dimensiones.

Un array de dos dimensiones equivale a una tabla con múltiples filas y múltiples columnas (Fig. 7.8).

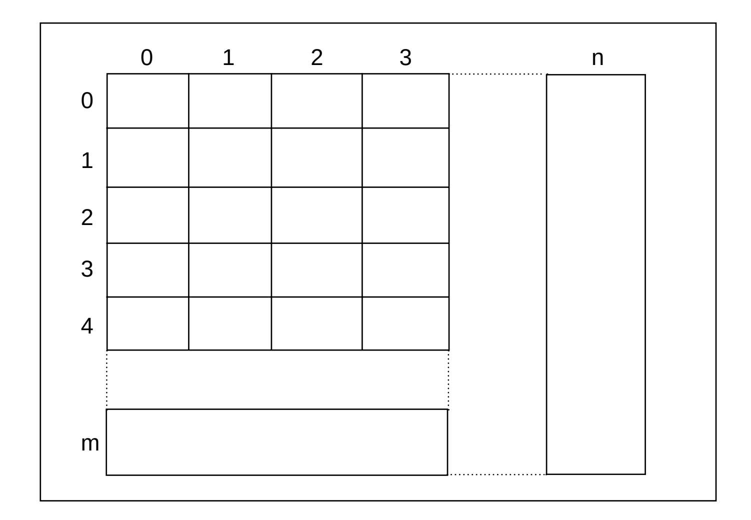


Figura 7.8. Estructura de un array de dos dimensiones.

Obsérvese que en el array bidimensional de la Figura 7.8, si las filas se etiquetan de *0* a *m* y las columnas de *0* a *n*, el número de elementos que tendrá el array será el resultado del producto *(m + 1)* × × *(n + 1)*. El sistema de localizar un elemento será por las coordenadas representadas por su número de fila y su número de columna *(a, b)*. La sintaxis para la declaración de un array de dos dimensiones es:

<tipo de datoElemento> <nombre array> [<NúmeroDeFilas<] [<NúmeroDeColumnas>]

Algunos ejemplos de declaración de tablas son:

```
char Pantalla[25][80];
int puestos[6][8];
int equipos[4][30];
int matriz[4][2];
```

Atención

Al contrario que otros lenguajes, C++ requiere que cada dimensión esté encerrada entre corchetes. La sentencia

```
int equipos[4, 30]
```

no es válida.

Un array de dos dimensiones en realidad es un *array de arrays*. Es decir, es un array unidimensional, y cada elemento no es un valor entero de coma flotante o carácter, sino que cada elemento es otro array.

Los elementos de los arrays se almacenan en memoria de modo que el subíndice más próximo al nombre del array es la fila y el otro subíndice, la columna. En la Tabla 7.1 se representan todos los elementos y sus posiciones relativas en memoria del array

```
int tabla[4] [2];
```

Tabla 7.1. Un array bidimensional.

Elemento	Posición relativa de memoria
`tabla[0][0]`	0
`tabla[0][1]`	2
`tabla[1][0]`	4
`tabla[1][1]`	6
`tabla[2][0]`	8
`tabla[2][1]`	10
`tabla[3][0]`	12
`tabla[3][1]`	14

7.4.1. Inicialización de arrays multidimensionales

Los arrays multidimensionales se pueden inicializar, al igual que los de una dimensión, cuando se declaran. La inicialización consta de una lista de constantes separadas por comas y encerradas entre llaves, como en

1.
```
int tabla1[2][3] =  {51, 52, 53, 54, 55, 56};
```

o bien en los formatos:

```
int tabla[2][3]=     {{51, 52, 53},
                      {54, 55, 56} };
int tabla[2][3]=     {{51, 52, 53}, {54, 55, 56}};
int tabla[2][3]=     {
                         {51, 52, 53}
                         {54, 55, 56}
                     };
```

2.
```
int tabla2[3][4] =  {
                         {1, 2, 3, 4},
                         {5, 6, 7, 8},
                         {9, 10, 11, 12}
                     };
```

Consejo

Los arrays multidimensionales (a menos que sean globales) no se inicializan a valores específicos a menos que se les asignen valores en el momento de la declaración o en el programa. Si se inicializan uno o más elementos, pero no todos, C++ rellena el resto con ceros o valores nulos (`'\0'`). Si se desea inicializar a cero un array multidimensional, utilice una sentencia tal como ésta.

```
float ventas[3][4] = {0.0};
```

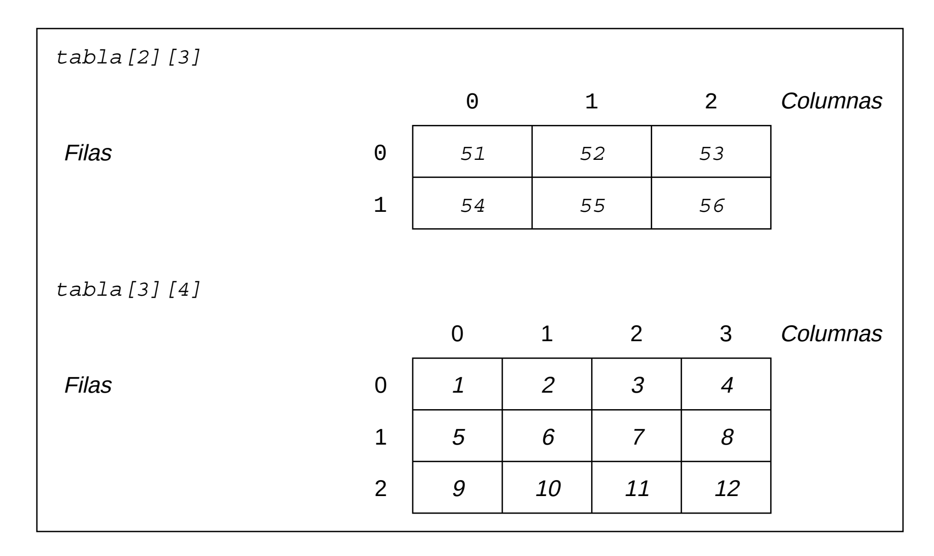


Figura 7.9. Tablas de dos dimensiones.

```
Tabla  [0][0]
       [0][1]
       [0][2]
       [0][3]
       [1][0]
       [1][1]
       [1][2]
       [1][3]
       [2][0]
       [2][1]
       [2][2]
       [2][3]
```

Figura 7.10. Almacenamiento en memoria de tabla [3][4].

7.4.2. Acceso a los elementos de arrays bidimensionales

Se puede acceder a los elementos de arrays bidimensionales de igual forma que a los elementos de un array unidimensional. La diferencia reside en que en los elementos bidimensionales deben especificarse los índices de la fila y la columna.

El formato general para asignación directa de valores a los elementos es:

```
inserción de elementos
  <nombre array>[índice fila][índice columna]=valor elemento;

extracción de elementos
  <variable> = <nombre array> [índice fila] [índice columna];
```

Algunos ejemplos de inserciones pueden ser:

```
Tabla[2][3] = 4.5;
Resistencias[2][4] = 50;
AsientosLibres[5][12] = 5;
```

y de extracción de valores:

```
Ventas = Tabla[1][1];
Día = Semana[3][6];
```

7.4.3. Lectura y escritura de elementos de arrays bidimensionales

Las sentencias `cin` y `cout` se utilizan para extraer elementos de arrays. Por ejemplo,

```
cin >> Tabla[2][3];
cout << Tabla[1][1];
cin >> Resistencias[2][4];
if (AsientosLibres[3][1])
   cout << "VERDADERO";
else
   cout << "FALSO";
```

7.4.4. Acceso a elementos mediante bucles

Se puede acceder a los elementos de arrays bidimensionales mediante bucles anidados. Su sintaxis es

```
for (int IndiceFila = 0; IndiceFila < NumFilas; ++IndiceFila)
   for (int IndiceCol = 0; IndiceCol < NumCol; ++IndiceCol)
      Procesar elemento[IndiceFila] [IndiceCol]
```

Ejemplo 7.6

```
// Rellenar la Tabla
float discos[2] [4];
int fila, col;
for (fila = 0; fila < 2; fila++) {
   for (col = 0; col < 4; col++)
   {
      cin >> discos[fila][col];
   }
}

// Visualizar la tabla
for (fila = 0; fila < 2; fila++) {
   for (col = 0; col < 4; col++)
   {
      cout << "Pts" << discos[fila] [col] << "\n";
   }
}
```

7.4.5. Arrays de más de dos dimensiones

C++ proporciona la posibilidad de almacenar varias dimensiones, aunque raramente los datos del mundo real requieren más de dos o tres dimensiones. El medio más fácil de dibujar un array de tres dimensiones es imaginar un cubo tal como se muestra en la Figura 7.11.

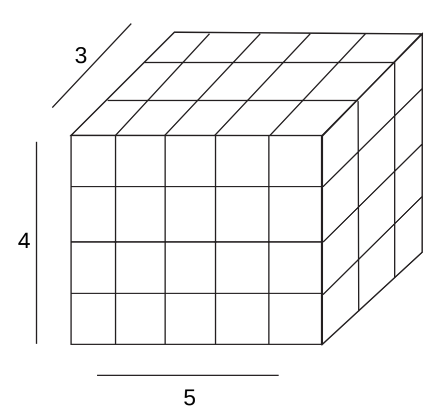


Figura 7.11. Un array de tres dimensiones (4 × 5 × 3).

Un array tridimensional se puede considerar como un conjunto de arrays bidimensionales combinados juntos para formar, en profundidad, una tercera dimensión. El cubo se construye con filas (dimensión vertical), columnas (dimensión horizontal) y planos (dimensión en profundidad). Por consiguiente, un elemento dado se localiza especificando su plano, fila y columna. Una definición de un array tridimensional equipos es:

```
int equipos[3][15][10];
```

Un ejemplo típico de un array de tres dimensiones es el modelo *libro*, en el que cada página del libro es un array bidimensional construido por filas y columnas. Así, por ejemplo, cada página tiene cuarenta y cinco líneas que forman las filas del array y ochenta caracteres por línea, que forman las columnas del array. Por consiguiente, si el libro tiene quinientas páginas, existirán quinientos planos y el número de elementos será 500 × 80 × 45 = 1.800.000.

7.4.6. Una aplicación práctica

El array `libro` tiene tres dimensiones `[PAGINAS] [LINEAS] [COLUMNAS],` que definen el tamaño del array. El tipo de datos del array es `char`, ya que los elementos son caracteres.

¿Cómo se puede acceder a la información del libro? El método más fácil es mediante bucles anidados. Dado que el libro se compone de un conjunto de páginas, el bucle más externo será el bucle de página; y el bucle de columnas el bucle más interno. Esto significa que el bucle de filas se insertará entre los bucles página y columna. El código siguiente permite procesar el array

```
for (int Pagina = 0; Pagina < PAGINAS; ++Pagina)
   for (int Linea = 0; Linea < LINEAS; ++Linea)
      for (int Columna = 0; Columna < COLUMNAS; ++Columna)
         <procesar Libro[Pagina][Linea][Columna]>
```

Ejercicio 7.1

Lectura y visualización de un array de dos dimensiones

La función `leer` lee un array (una tabla) de dos dimensiones y la función `visualizar` presenta la tabla en la pantalla.

```
// prototipos
void leer(int a[][5]);
```

```
void visualizar(const int a[][5]);

main()
{
   int a[3][5];
   leer(a);
   visualizar(a);
}

void leer(int a[][5]
{
   cout << "Introduzca 15 números enteros, 3 por fila" << endl;
   for (int i = 0; i < 3; i++) {
      cout << "Fila " << i << ":";
      for (int j = 0; j < 5; j++)
         cin >> a[i][j];
   }
}

void visualizar (const int a[][5]
{
   for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 5; j++)
         cout << " " << a[i][j];
      cout << endl;
   }
}
```

Ejecución

La Tabla (array de dos dimensiones)

```
Fila 0:      45 75 25 10 40
Fila 1:      20 14 36 15 26
Fila 2:      21 15 37 16 27
```

Y la traza (ejecución) del programa es:

```
   Introduzca 15 números enteros, 3 por fila

Fila 0: 45 75 25 10 40
Fila 1: 20 14 36 15 26
Fila 2: 21 15 37 16 27
45 75 25 10 40
20 14 36 15 26
21 15 37 16 27
```

7.5. UTILIZACIÓN DE ARRAYS COMO PARÁMETROS

En C++ *todos los arrays se pasan por referencia* (dirección). Esto significa que cuando se llama a una función y se utiliza un array como parámetro, se debe tener cuidado de no modificar los arrays en una

función llamada. C++ trata automáticamente la llamada a la función como si hubiera situado el operador de dirección & delante del nombre del array. La Figura 7.12 ayuda a comprender el mecanismo.

```
main()
{
  char palabra[4]= "ABC"
  cambiar(palabra);
  cout << palabra << endl;
  return;
}
```

palabra

```
cambiar(char c[4])
{
   cout << c << endl;
   strcpy(c, "AMA");
   return;
}
```

Figura 7.12. Paso de un array por dirección.

Dadas las declaraciones

```
const MAX 100
double datos[MAX];
```

se puede declarar una función que acepte un array de valores double como parámetro. Se puede prototipar una función `SumaDeDatos`, de modo similar a:

```
double SumaDeDatos(double datos[MAX]);
```

Incluso mejor si se dejan los corchetes en blanco y se añade un segundo parámetro que indica el tamaño del array:

```
double SumaDeDatos(double datos[], int n);
```

A la función `SumaDeDatos` se pueden entonces pasar argumentos de tipo array junto con un entero `n`, que informa a la función sobre cuántos valores contiene el array. Por ejemplo, esta sentencia visualiza la suma de valores de los datos del array:

```
cout << "Suma de " << SumaDeDatos(datos, MAX);
```

La función `SumaDeDatos` no es difícil de escribir. Un simple bucle for suma los elementos del array y una sentencia `return` devuelve el resultado de nuevo al llamador:

```
double SumaDeDatos(double datos[], int n)
{
   double Suma = 0;

   while (n > 0)
      Suma += datos[n];

   return Suma;
}
```

El código que se utiliza para pasar un array a una función incluye el tipo de elemento del array y su nombre. El siguiente ejemplo incluye dos funciones que procesan arrays. En ambas listas de parámetros, el array `a[]` se declara en la lista de parámetros tal como

```
double a[]
```

El número real de elementos se pasan mediante una variable entera independiente. Cuando se pasa un array a una función, se pasa realmente *sólo* la dirección de la celda de memoria donde comienza el array. Este valor se representa por el nombre del array `a`. La función puede cambiar entonces el contenido del array accediendo directamente a las celdas de memoria en donde se almacenan los elementos del array. Así, aunque el nombre del array se pasa por valor, sus elementos se pueden cambiar como si se hubieran pasado por referencia.

Ejemplo 7.7

Paso de arrays a funciones

```
const int long = 100;
void leerArray(double [], int&);
void imprimirArray (const double [], const int);

main()
{
   double a[long];
   int n;
   leerArray(a, n);
   cout << "El array tiene " << n << " elementos\n Son:\n";
   imprimirArray(a, n);
}

void leerArray(double a[], int& n)
{
   n = 0;
   cout << "Introduzca datos. Para terminar pulsar 0:\n";
   for (n = 0; n < long; n++)
   {
      cout << n << ":";
      cin >> a[n];
      if (a[n] == 0) break;
   };
}

void imprimirArray(const double a[], const int n)
{
   for (int i = 0; i < n; i++)
      cout << "\t" << i << ":" << a[i] << endl;
}
```

Ejecución

```
Introduzca datos. Para terminar pulsar 0.
0:    31.31
1:    15.25
```

```
2:    44.77
3:    0
El array tiene 3 elementos
son:
0:    31.31
1:    15.25
2:    44.77
```

Ejercicio 7.2

Escribir una función que sume los primeros n elementos de un array especificado.

```
double suma(const double a[], const int n)
{
   double s = 0.0;
   for (int i = 0; i < n; i++)
      s += a[i];
   return s  ;
}
```

7.5.1. Precauciones

Cuando se utiliza una variable array como argumento, la función receptora puede no conocer cuántos elementos existen en el array. Sin su conocimiento una función no puede utilizar el array. Aunque la variable array puede apuntar al comienzo del array, no proporciona ninguna indicación de donde termina el array.

Una función `SumaDeEnteros` suma los valores de todos los elementos de un array y devuelve el total.

```
int SumaDeEnteros(int *ArrayEnteros)
{
   //...
}

main()
{
   int Lista[5] = {10, 11, 12, 13, 14};
   SumaDeEnteros (Lista);
   //...
}
```

Aunque `SumaDeEnteros()` conoce dónde comienza el array, no conoce cuántos elementos hay en el array; en consecuencia, no sabe cuántos elementos hay que sumar.

Se pueden utilizar dos métodos alternativos para permitir que una función conozca el número de argumentos asociados con un array que se pasa como argumento de una función:

- situar un valor de señal al final del array, que indique a la función que se ha de detener el proceso en ese momento;
- pasar un segundo argumento que indica el número de elementos del array.

Todas las cadenas terminadas en nulo utilizan el primer método. Una segunda alternativa es pasar el número de elementos del array siempre que se pasa el array como un argumento. El array y el número de elementos se convierten entonces en una pareja de argumentos que se asocian con la función llamada. La función `SumaDeEnteros()`, por ejemplo, se puede actualizar así:

```
int SumaDeEnteros(int *ArrayEnteros, int NoElementos)
{
   //...
}
```

El segundo argumento, `NoElementos`, es un valor entero que indica a la función `SumaDeEnteros` cuántos elementos se procesarán en el array `Array Enteros.`

Este método suele ser el utilizado para arrays de elementos que no son caracteres.

Ejemplo 7.8

```
#include <iostream>
using namespace std;

int SumaDeEnteros(int *ArrayEnteros, int NoElementos)
{
   int i, Total = 0;
   for (i = 0; i < NoElementos; i++)
      Total += ArrayEnteros[i];

   return Total;
}

void main()
{
   int Items[10];
   int Total, i;
   cout << "Introduzca 10 números, seguidos por return" << endl;
   for (i = 0; i < 10; i++)
      cin >> Items[i];

   // Suma del array
   Total = SumaDeEnteros(Items, 10);

   // Visualizar el resultado
   cout << Total;
   return 0;
}
```

El siguiente programa muestra cómo se pasa un array de enteros a una función de ordenación, `ordenar`.

```
void main()
{
   int ListaEnt[ ] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 10};
   int LongLista = sizeof(ListaEnt) / sizeof(int);

   void ordenar(int *, int);    //prototipo de ordenar
```

```
   for (unsigned i = 0; i < LongLista; i++)
      cout << ListaEnt[i] << " ";
   return 0;
}

void ordenar(int *lista, int numElementos)
{
   //cuerpo de la función ordenar el array
}
```

Como C++ trata las cadenas como arrays de caracteres, las reglas para pasar arrays como argumentos a funciones se aplican también a cadenas. El siguiente ejemplo de una función de cadena que convierte los caracteres de sus argumentos a mayúsculas, muestra el paso de parámetros tipo cadena.

7.5.2. Paso de cadenas como parámetros

La técnica de pasar arrays como parámetros se utiliza para pasar cadenas de caracteres a funciones. Las cadenas terminadas en nulo utilizan el primer método dado anteriormente para controlar el tamaño de un array. Las cadenas son arrays de caracteres. Cuando una cadena se pasa a una función, tal como `strlen()` (*véase* Capítulo 11), la función conoce que se ha almacenado el final del array cuando ve un valor de 0 en un elemento del array.

Las cadenas utilizan siempre un 0 para indicar que es el último elemento del array de caracteres, pero 0 no es el único indicador de final de cadena que se puede utilizar. Se puede utilizar cualquier valor para indicar el elemento final de sus arrays.

Consideremos estas declaraciones de una constante y una función que acepta un parámetro cadena y un valor de su longitud.

```
const MAXLON = 128;
void FuncDemo(char s[], int long);
```

El parámetro `s` es un array de caracteres de longitud no especificada. El parámetro `long` indica a la función cuántos bytes ocupa `s` (que puede ser diferente del número de caracteres almacenados en `s`). Dadas las declaraciones siguientes:

```
char presidente[MAXLON] = "Manuel Martínez";
FuncDemo(presidente, MAXLON);
```

la primera línea declara e inicializa un array de caracteres llamado `presidente`, capaz de almacenar hasta `MAXLON-1` caracteres más un byte de terminación nulo. La segunda línea pasa la cadena a la función.

7.6. ORDENACIÓN DE LISTAS

La *ordenación* de arrays es otra de las tareas usuales en la mayoría de los programas. La ordenación o clasificación es el procedimiento mediante el cual se disponen los elementos del array en un orden especificado, tal como orden alfabético u orden numérico.

Un diccionario es un ejemplo de una lista ordenada alfabéticamente, y una agenda telefónica o lista de cuentas de un banco es un ejemplo de una lista ordenada numéricamente. El orden de clasificación u ordenación puede ser *ascendente* o *descendente*.

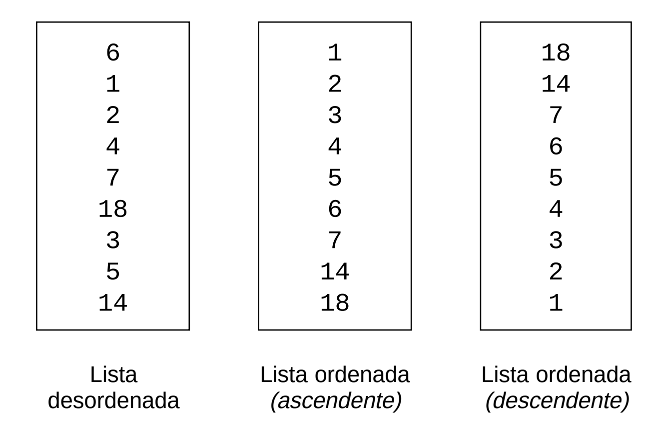


Figura 7.13. Lista de números desordenada y ordenada en orden ascendente y en orden descendente.

Existen numerosos algoritmos de ordenación de arrays: ***inserción, burbuja, selección, rápido*** *(quick sort)*, ***fusión*** *(merge)*, ***montículo*** *(heap)*, ***shell***, etc. En los Capítulos 12 y 20 se realizará un estudio más en profundidad sobre ordenación.

7.6.1. Algoritmo de la burbuja

La ordenación por burbuja es uno de los métodos más fáciles de ordenación. El método (algoritmo) de ordenación es muy simple. Se compara cada elemento del array con el siguiente (por parejas), si no están en el orden correcto, se intercambian entre sí sus valores. El valor más pequeño *flota* hasta la parte superior del array como si fuera una burbuja en un vaso de refresco con gas (soda, sifón, etc.).

La Figura 7.14 muestra una lista de números, antes, durante las sucesivas comparaciones y a la terminación del algoritmo de la burbuja. Se van realizando diferentes pasadas hasta que la lista se encuentra ordenada totalmente en orden ascendente.

Lista desordenada:	6	4	10	2	8
Primera pasada	6	4	4	4	
	4	6	6	6	
	10	10	2	2	
	2	2	10	8	
	8	8	8	10	
Segunda pasada	4	4			
	6	2			
	2	6			
	8	8			
	10	10			
Tercera pasada	4	2			
	2	4			
	6	6			
	8	8			
	10	10			
Cuarta pasada	2				
	4				
	6				
	8				
	10				

Figura 7.14. Pasadas en el algoritmo de burbuja.

La ordenación de arrays requiere siempre un intercambio de valores, cuando éstos no se encuentran en el orden previsto. Si, por ejemplo, en la primera pasada 6 y 4 no están ordenados se han de intercambiar sus valores. Suponiendo que el array se denomina lista:

```
lista[0]   6
lista[1]   4
lista[2]   10
lista[3]   2
lista[4]   8
```

para intercambiar dos valores, se necesita utilizar una tercera variable auxiliar que contenga el resultado inmediato. Así, por ejemplo, si las dos variables son `lista[0]` y `lista[1]`, el siguiente código realiza el intercambio de dos variables:

```
aux      = lista[0];
lista[0] = lista[1];
lista[1] = aux;
```

Ejemplo 7.9

La función `intercambio` *intercambia los valores de dos variables x e y.*

El algoritmo de intercambio utiliza una variable auxiliar `aux`

```
aux = x;
x   = y;
y   = aux;
```

La función `intercambio` sirve para intercambiar los dos objetos `x` e `y` que se pasan a ella.

```
void intercambio(float& x, float& y)
{
   float aux = x;
   x = y;
   y = aux;
}
```

Ejemplo 7.10

El programa siguiente ordena una lista de números reales y a continuación los imprime.

```
// prototipos
void imprimir(float [], const int);
void ordenar(float [], const int);

void main()
{
   float a[10]={25.5,34.1,27.6,15.24. 3.27, 5.14, 6.21, 7.57, 4.61, 5.4};
   imprimir(a, 10);
   ordenar(a, 10);
   imprimir(a, 10);
```

```
}

void imprimir(float a[], const int n)
{
   for (int i = 0; i < n; i++) {
      cout << a[i] << ", ";
      if ((i + 1)% 16 == 0) cout << endl;
    }
   cout << endl;
}

void intercambio(float& x, float& y);

// ordenar burbuja
void ordenar(float a[], const int n)
{
   for (int i = n; i < 0; i--)
      for (int j = 0; j < i; j++)
         if (a[j] > a[j+1]) intercambio(a[j], a[j+1]);
}
```

La función `OrdenarBur` realiza una ordenación por el método de la burbuja del array Lista, que es el parámetro de entrada/salida.

```
// ORDENBUR.CPP
void burbuja(int lista[], int n)
// lista[], parámetro de entrada/salida
// n, parámetro de entrada, número de elementos

{
   int i,      //
      paso,    //número de pasadas a través del array
      aux,     //variable auxiliar utilizada en el intercambio
      ordenado;//variable que implica si el orden es correcto

   paso = 1;
   do {
      // se supone que el array está ordenado hasta que
      // se encuentra una pareja desordenada
      ordenado = 1;
      //Realiza una pasada a través de elementos desordenados
      for ( i = 0; i < n-paso; ++i)
      {
         if (lista[i] > lista[i+1])
         {
            // Intercambio de parejas de variables
            // desordenadas
            aux        = lista[i];
            lista[i]   = lista[i+1];
            lista[i+1] = aux;
            ordenado = 0;
         }
```

```
        }
        ++paso;
    } while (!ordenado);
}
```

7.7. BÚSQUEDA EN LISTAS

Los *arrays* (listas y tablas) son uno de los medios principales por los cuales se almacenan los datos en programas C++. Debido a esta causa, existen operaciones fundamentales cuyo tratamiento es imprescindible conocer. Estas operaciones esenciales son: la *búsqueda* de elementos y la ordenación o clasificación de las listas.

La *búsqueda* de un elemento dado en un array (lista o tabla) es una aplicación muy usual en el desarrollo de programas en C++. Dos algoritmos típicos que realizan esta tarea son la ***búsqueda secuencial*** o ***en serie*** y la ***búsqueda binaria*** o ***dicotómica***. La búsqueda secuencial es el método utilizado para listas no ordenadas, mientras que la búsqueda binaria se utiliza en arrays que ya están ordenados. En los Capítulos 12 y 20 se profundizará en el estudio de los algoritmos de búsqueda.

7.7.1. Búsqueda secuencial

Este algoritmo busca el elemento dado, recorriendo secuencialmente el array desde un elemento al siguiente, comenzando en la primera posición del array y se detiene cuando se encuentra el elemento buscado o bien se alcanza el final del array.

Por consiguiente, el algoritmo debe comprobar primero el elemento almacenado en la primera posición del array, a continuación el segundo elemento, y así sucesivamente, hasta que se encuentra el elemento buscado o se termina el recorrido del array. Esta tarea repetitiva se realizará con bucles, en nuestro caso con el bucle **`while`**.

Algoritmo *`BusquedaSec`*

Se utiliza una variable lógica (*booleana*) denominada *`Encontrado`*, que indica si el elemento se encontró en la búsqueda. La variable `Encontrado` se inicializa a *falso* y se activa a *verdadero* cuando se encuentra el elemento. Se utiliza un operador **and**, que permita evaluar las dos condiciones de terminación de la búsqueda: elemento encontrado o terminada la búsqueda (índice del array excede al último valor válido del mismo).

Cuando el bucle se termina, el elemento o bien se ha encontrado, o bien no se ha encontrado. Si el elemento se ha encontrado, el valor de *`Encontrado`* será *verdadero* y el valor del índice será la posición del array (índice del elemento encontrado). Si el elemento no se ha encontrado, el valor de *`Encontrado`* será *falso* y se devuelve el valor –1 al programa llamador.

`BusquedaSec`

```
inicio
   Poner Encontrado = falso
   Poner Indice = primer indice del array
   mientras (Elemento no sea Encontrado) y (Indice < Ultimo
            indice) hacer
      si (A[indice] == Elemento) entonces
         Poner Encontrado a Verdadero
      si no
         Incrementar Indice
```

```
      si (Encontrado == Verdadero) entonces
         retorno (Indice)
fin_mientras
      si no
         retorno (-1)
fin
```

El algoritmo anterior implementado como una función para un array `Lista` es:

```
int BusquedaSec(int Lista[MAX], int Elemento)
{
   enum {FALSE, TRUE}
   int ENCONTRADO = FALSE;
   int i = 0;

   // Búsqueda en la lista hasta que se encuentra el elemento
   // o se alcanza el final de la lista
   while ((!Encontrado) && (i <= MAX-1))
   {
      if (A[i] == Elemento)
         Encontrado = TRUE;
      else
         ++i;
   }
   // Si se encuentra el elemento
   // Se devuelve la posición en la lista (return -1)
   if (Encontrado)
      return (i);
   else
      return (-1);
}
```

Ejemplo 7.11

El siguiente programa busca un elemento y la posición que ocupa en la lista o array.

```
void buscar(int& encontrado, int& posicion, int a[], int n,
            int destino);

void main()
{
   int a[]= {25, 24, 65, 99, 77, 44, 33, 88, 66}, destino,
             encontrado, pos};
   do {
      cout << "Destino:";
      cin >> destino;
      buscar(encontrado, pos, a, 9 destino);
      if(encontrado) cout << destino << "está en a[" << pos << "].\n";
      else cout << destino << "no se encuentra.\n";
   } while (destino != 0);
}
```

```
// Búsqueda lineal:
void buscar(int &encontrado, int &posicion, int a[], int n, int destino)
{
   encontrado = posición = 0;
   while (!encontrado && posición < n)
      encontrado = (a[posición++] == destino);
   --posición;
}
```

Ejecución

```
Destino: 65
65 está en a[2]
Destino: 77
77 está en a[4]
Destino: 0
0 no se encuentra
```

Ejemplo 7.12

Encontrar el número mayor de una lista.

```
// Nombre del archivo BUSSEC1.CPP
// Encontrar el mayor valor del array

#include <iostream>
using namespace std;

#define LONG 15

void main()
{
   int lista[LONG] = { 5, 2, 4, 9, 14,
                       4, 2, 85, 11, 45,
                       25, 12, 45, 6, 99 };
   int valor_mayor, i;

   valor_mayor = lista[0];

   for (i = 1; i < LONG; i++)
   {
      // Almacenar valor actual si es mayor que
      // el último valor mayor encontrado
      if (lista[i] > valor_mayor)
      {
         valor_mayor = lista[i];
      }
   }
   cout  << "El número mayor de la lista es"
         << valor_mayor << "\n";
   return 0;
}
```

RESUMEN

En este capítulo se analizan los tipos agregados de C++ «*arrays*» o **arreglos**. Después de leer este capítulo debe tener un buen conocimiento de los conceptos fundamentales de los tipos agregados.

Se describen y analizan los siguientes conceptos:

- Un *array* (**arreglo**) es un tipo de dato estructurado que se utiliza para localizar y almacenar elementos de un tipo de dato dado.
- Existen arrays de una dimensión, de dos dimensiones... y multidimensionales.
- En C++ los arrays se definen especificando el tipo de dato del elemento, el nombre del array y el tamaño de cada dimensión del array. Para acceder a los elementos del array se deben utilizar sentencias de asignación directas, sentencias de lectura/escritura o bucles (mediante las sentencias for, while o do-while).

```
int total_meses[12];
```

- Los arrays de caracteres contienen cadenas de textos. En C++ se terminan las cadenas de caracteres situando un carácter nulo ('\0') como último byte de la cadena.
- C++ soporta arrays multidimensionales.

```
int ventas_totales[12] [50];
```

EJERCICIOS

7.1. Escribir un programa que lea una secuencia de números en un orden dado: *Aplicación.* Leer 4 números y, a continuación, visualizarlos en orden directo y luego en orden inverso.

7.2. Inicializar un array. *Ejemplo.* Un *array* de 4 elelementos reales `(double)` de valores: 22.5, 44.5, 77.5, 99.5.

7.3. Escribir un programa que inicialice una lista de 4 elementos, pero sólo se proporcionan 2. ¿Cuál será el valor de los elementos sin asignar?

7.4. Calcular la suma de los *n* elementos de un array especificado.

7.5. Calcular la suma de los *n* primeros elementos de un array especificado.

7.6. Se tiene una lista de números enteros almacenados en un array. Se pide ordenarlos: (1) en orden creciente; (2) en orden decreciente.

7.7. Se dispone de una lista de las temperaturas medias de una ciudad en cada uno de los días de una determinada semana. Se desea calcular la temperatura mayor y menor.

7.8. Se desea introducir por teclado una tabla de 4 filas por 5 columnas de elementos enteros. Escribir un programa que tras la lectura los visualice en la pantalla.

7.9. Leer 5 valores de una lista de números enteros y deducir cuál es la distancia que les separa del número mayor.

7.10. ¿Cuál es la salida del código siguiente? (Se supone insertado en un programa completo y correcto.)

```
char simbolo[3] = {'a', 'b', 'c',
                  'd'} ;
cout a[0] << " " << a[1] << " "
          << a[2] << endl ;
     a[1] = a[2] ;
     cout << a[0] << " " << a[1]
          << " " << a[2] << endl ;
```

7.11. ¿Cuál es la salida del siguiente código?

```
char simbolo[3] = {'a', 'b', 'c'} ;
for (int indice=0 ; indice<3 ; in-
     dice++)
      cout << simbolo[indice] ;
```

7.12. ¿Cuál es la salida del siguiente código?

```
int i, aux[10] ;
for (i=0 ; i<10 ; i++)
      aux[i] = 2*i ;
for (i=0 ; i<10 ; i++)
      cout << aux[i] << " " ;
cout << endl ;
for (i=0 ; i<10 ; i = i + 2)
cout << aux[i] << " " ;
```

7.13. Considere la siguiente definición:

```
void triple (int &n)
```

```
{
  n = 3*n ;
}
```

¿Cuál de las siguientes funciones es válida?

```
int a[3] = {4, 5, 6}, numero = 2 ;
triple (numero) ;
triple (a[2]) ;
triple (a[3]) ;
triple [a[numero]) ;
triple (a) ;
```

7.14. Considere la siguiente definición de función:

```
void  prueba2(int a[ ], int numero)
 {
  for (int indice = 0-; indice <
       numero-; indice++)
             a[indice] = 2 ;
}
```

¿Cuál de las siguientes funciones es válida?

```
int mi_array[30] ;
prueba2(mi_aray, 30) ;
prueba2(mi_array, 10) ;
prueba3(mi_array, 40) ;
int tu_array[100] ;
prueba2(tu_array, 100) ;
prueba2(tu_array[3], 30) ;
```

PROBLEMAS

7.1. Escribir un programa que convierta un número romano (en forma de cadena de caracteres) en número arábigo.

Reglas de conversión

M	1000
D	500
C	100
L	50
X	10
V	5
I	1

7.2. Escribir una función que invierta el contenido de *n* números enteros. El primero se vuelve el último; el segundo, el penúltimo, etc.

7.3. Escribir una función a la cual se le proporcione una fecha (Día, Mes, Año), así como un número de días a añadir a esta fecha. La función calcula la nueva fecha y se visualiza.

7.4. Un número entero es primo si ningún otro número primo más pequeño que él es divisor suyo. A continuación, escribir un programa que rellene una tabla con los 80 primeros números primos y los visualice.

7.5. El juego del ahorcado se juega con dos personas (o una persona y una computadora). Un jugador selecciona una palabra y el otro jugador trata de adivinar la palabra adivinando letras individuales. Diseñar un programa para jugar al ahorcado. *Sugerencia*: almacenar una lista de palabras en un array y seleccionar palabras aleatoriamente.

7.6. Escribir un programa que lea las dimensiones de una matriz, lea y visualice la matriz y, a continuación, encuentre el mayor y menor elemento de la matriz y sus posibilidades.

7.7. Si x representa la media de los números $x1$, $x2$,... xn, entonces la *varianza* es la media de los cuadrados de las desviaciones de los números de la media.

$$Varianza \ = \frac{1}{n} \sum_{i=1}^{n} (x^1 - x^2)^2$$

y la *desviación estándar* es la raíz cuadrada de la varianza. Escribir un programa que lea una lista de números reales, los cuente y, a continuación, calcule e imprima su media, varianza y desviación estándar. Utilizar funciones para calcular la media, varianza y desviación estándar.

7.8. Escribir una función que acepte como parámetro un vector que puede contener elementos duplicados. La función debe sustituir cada valor repetido por –5 y devolver al punto donde fue

llamado el vector modificado y el número de entradas modificadas.

7.9. Los resultados de las últimas elecciones a alcalde en el pueblo *x* han sido los siguientes:

Distrito	*Candidato A*	*Candidato B*	*Candidato C*	*Candidato D*
1	194	48	206	45
2	180	20	320	16
3	221	90	140	20
4	432	50	821	14
5	820	61	946	18

Escribir un programa que haga las siguientes tareas:

a) Imprimir la tabla anterior con cabeceras incluidas.
b) Calcular e imprimir el número total de votos recibidos por cada candidato y el porcentaje del total de votos emitidos. Asimismo, visualizar el candidato más votado.
c) Si algún candidato recibe más del 50 por 100 de los datos, el programa imprimirá un mensaje declarándole ganador.
d) Si ningún candidato recibe más del 50 por 100 de los datos, el programa debe imprimir el nombre de los dos candidatos más votados, que serán los que pasen a la segunda ronda de las elecciones.

7.10. Escribir un programa que lea una colección de cadenas de caracteres de longitud arbitraria. Por cada cadena leída, su programa hará lo siguiente:

a) Imprimir la longitud de la cadena.
b) Contar el número de ocurrencia de palabras de cuatro letras.
c) Sustituir cada palabra de cuatro letras por una cadena de cuatro asteriscos e imprimir la nueva cadena.

7.11. Una agencia de venta de vehículos automóviles distribuye quince modelos diferentes y tiene en su plantilla diez vendedores. Se desea un programa que escriba un informe mensual de las ventas por vendedor y modelo, así como el número de automóviles vendidos por cada vendedor y el número total de cada modelo vendido por todos los vendedores. Asimismo, para entregar el premio al mejor vendedor, necesita saber cuál es el vendedor que más coches ha vendido.

vendedor \ modelo	1	2	3	4	15
1	4	8	1		4
2	12	4	25		14
3	15	3	4		7
.					
.					
10					

7.12. En un tablero de ajedrez la reina puede atacar cualquier pieza que esté en la misma fila, columna o diagonal que la reina. El problema de las *n* reinas es posicionar *n* reinas en un tablero $n \times n$, de modo que ninguna reina pueda atacar a ninguna otra. Escribir un programa que solucione este problema para un valor dado de *n*.

7.13. Escribir un programa que transforme un número arábigo en romano (cadena de caracteres en número arábigo). Los valores de las cifras romanas son los siguientes:

M	1000
D	500
C	100
L	50
X	10
V	5
I	1

7.14. Escribir una función que invierta el orden de *n* en números enteros. El primero se pone en última posición, el segundo en penúltima, etc.

7.15. Calcular la media aritmética de una lista de números reales.

7.16. Calcular el mayor de una lista de números.

7.17. Diseñar un programa que determine la frecuencia de aparición de cada letra mayúscula en un texto escrito por el usuario (fin de lectura, el punto o el retorno de carro, ASCII 13).

7.18. Escribir un programa que lea una cadena de caracteres y la visualice en un cuadro.

7.19. Escribir un programa que lea una frase, sustituya todas las secuencias de dos o más blancos por un solo blanco y visualice la frase restante.

7.20. Escribir un programa que lea una frase y, a continuación, visualice cada palabra de la frase en

columna, seguido del número de letras que compone cada palabra.

7.21. Escribir un programa que desplace una palabra leída del teclado desde la izquierda hasta la derecha de la pantalla.

7.22. Escribir un programa que visualice en forma codificada la cadena de caracteres leída del teclado.

7.23. Escribir un programa que calcule la frecuencia de aparición de las vocales de un texto proporcionado por el usuario. Esta solución debe presentarse en forma de histograma.

7.24. Escribir un programa que lea una serie de cadenas, a continuación determine si la cadena es un identificador válido C++. *Sugerencias*: utilizar los siguientes subprogramas: longitud (tamaño del identificador en el rango permitido); primero (determinar si el nombre comienza con un símbolo permitido); restantes (comprueba si los restantes son caracteres permitidos).

7.25. Escribir una función conversión que reciba como parámetro una cadena representando una fecha en formato 'dd/mm/aa', como 17/11/91, y la devuelva en formato de texto: 17 noviembre 1991.

7.26. Escriba una función **`sort`** que ordene un conjunto de *n* cadenas en orden alfabético.

7.27. Diseñar un programa que determine la media del número de horas trabajadas durante todos los días de la semana.

7.28. Escriba una función que ordene los *n* primeros elementos de un array de cadenas basado en las longitudes de las cadenas. Por ejemplo, 'bibi' vendrá después que 'Ana'.

7.29. Se introduce una frase por teclado. Se desea imprimir cada palabra de la frase en líneas diferentes y consecutivas.

7.30. Escribir un programa que permita escribir en sentido inverso una cadena de caracteres.

7.31. Buscar una palabra en una cadena y calcular su frecuencia de aparición.

EJERCICIOS RESUELTOS

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 149).

2. Sitio web del libro, *`www.mhe.es/joyanes`* (código fuente).

7.1. ¿Cuál es la salida del siguiente programa?

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
   int i, Primero[21];

   for i = 1; i <= 6; i++)
      cin >> Primero[i];
   for(i = 3; i > 0; i--)
     cout << Primero[2 * i];
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

7.2. ¿Cuál es la salida del siguiente programa?

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int i,j,k, Primero[10];

    for(i = 0; i < 10; i++)
         Primero[i]  =  i  +  3;
    cin >> j >> k;
    for(i = j; i <= k; i++)
        cout << Primero[i] << " ";
```

```
        system("PAUSE");
        return EXIT_SUCCESS;
    }
```

7.3. ¿Cuál es la salida del siguiente programa?

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int i, j ,k, Primero[11], Se-
        gundo[11];

    for(i = 0; i < 11;  i++)
      Primero[i] = 2 * i + 2;
    for(j = 0; j < 6; j++)
       Segundo[j] = Primero[2 * j]
           + j;
    for(k = 3; k < 6; k++)
       cout << Primero[k + 1]<< " "
            << Segundo [k – 1]
            << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

7.4. Escribir una función que rellene un array con los 10 primeros números impares, y visualice los contenidos del vector cuyos índices son: 0, 2, 4, 6, 8.

PROBLEMAS RESUELTOS

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 151).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

7.1. Escribir un programa que lea 10 números enteros, los almacene en un vector, y calcule la suma de todos ellos, así como su media aritmética.

7.2. Un vector se dice que es simétrico si el elemento que ocupa la posición *i*_ésima coincide con el que ocupa la posición *n* – *i*_ésima, siempre que el número de elementos que almacene el vector sea n. Por ejemplo el vector que almacena los valores 2, 4, 5, 4, 2 es simétrico. Escribir una función que decida si el vector de n datos que recibe como parámetro es simétrico.

7.3. Un vector que almacena `n` datos se dice que es mayoritario, si existe un elemento almacenado en el vector que se aparece en el vector más de `n / 2` veces. Escribir una función que decida si un vector es mayoritario.

7.4. Diseñar funciones que calculen el producto escalar de dos vectores, la norma de un vector y el coseno del ángulo que forman definidos de la siguiente forma:

$$pe(a,b,n) = \sum_{i=0}^{n-1} a(i) \cdot b(i)$$

$$norma(a,n) = \sqrt{\sum_{i=0}^{n-1} a(i) \cdot a(i)}$$

$$\cos(a,b,n) = \frac{\sum_{i=0}^{n-1} a(i) \cdot b(i)}{\sqrt{\sum_{i=0}^{n-1} a(i) \cdot a(i) \cdot \sum_{i=0}^{n-1} a(i) \cdot b(i)}}$$

7.5. Escribir un algoritmo que calcule y escriba una tabla con los 100 primeros números primos. Un número es primo si sólo tiene por divisores la unidad y el propio número.

7.6. Escribir un programa que genere aleatoriamente los datos de un vector, lo visualice, y calcule su media m, su desviación media dm su desviación típica dt, dadas por las siguientes expresiones:

$$m = \left(\sum_{i=0}^{n-1} a(i)\right) / n \qquad dm = \frac{\sum_{i=0}^{n-1} abs(a(i) - m)}{n}$$

$$dt = \sqrt{\frac{\sum_{i=0}^{n-1}(a(i) - m)^2}{n}}$$

7.7. Escribir una función que reciba como parámetro una matriz cuadrada de orden *n*, y calcule la traspuesta de la matriz almacenando el resultado en la propia matriz.

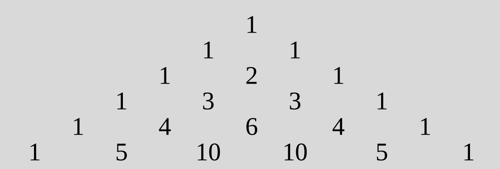
7.8. Se dice que una matriz tiene un punto de silla si alguna posición de la matriz es el menor valor de su fila, y a la vez el mayor de su columna. Escribir una función que tenga como parámetro una matriz de números reales, y calcule y escriba los puntos de silla que tenga, así como las posiciones correspondientes.

7.9. Escribir un programa que lea un número natural impar *n* menor o igual que 11, calcule y visualice el cuadrado mágico de orden *n*. Un cuadrado de orden `n*n` se dice que es mágico si contiene los valores 1, 2, 3, ..., *n*n*, y cumple la condición de que la suma de los valores almacenados en cada fila y columna coincide.

7.10. Escribir una función que encuentre el elemento mayor y menor de una matriz, así como las posiciones que ocupa y los visualice en pantalla.

7.11. Escribir una función que reciba como parámetro una matriz cuadrada de orden *n* y decida si es simétrica. Una matriz cuadrada de orden *n* es simétrica si *a[i][j]== a[j][i]* para todos los valores de los índices *i*, *j*.

7.12. Codificar un programa C++ que permita visualizar el triángulo de Pascal:



En el triángulo de Pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando primeramente un array bidimensional y, posteriormente, un array de una sola dimensión.

7.13. Escribir un programa que lea un texto de la entrada y cuente: el número de palabras leídas; número de líneas y vocales (letras *a*; letras *e*; letras *i*; letras *o*; letras *u*). El final de la entrada de datos viene dado por Control + Z.

7.14. Codificar un programa C++ que lea una frase, y decida si es palíndroma. Una frase se dice que es palíndroma si después de haber eliminado los blancos, se puede leer de igual forma en los dos sentidos.

7.15. Escribir una función que reciba un número escrito en una cierta base *b* como cadena de caracteres y lo transforme en el mismo número escrito en otra cierta base *b*1 como cadena de caracteres. Las bases *b* y *b*1 son mayores o iguales que 2 y menores o iguales que 16. Puede suponer que el número cabe en una variable numérica entero largo, y usar la base 10 como base intermedia.

7.16. Diseñar y codificar un programa que lea un texto y determine la frecuencia de aparición de cada letra mayúscula. El fin de lectura viene dado por (Control + Z).

7.17. Escribir un programa que lea una frase y, a continuación, visualice cada palabra de la frase en columna, seguido del número de letras que compone cada palabra.

CAPÍTULO 8

Estructuras y uniones

Contenido

INTRODUCCIÓN

Este capítulo examina las estructuras, uniones, enumeraciones y tipos definidos por el usuario que permiten a un programador crear nuevos tipos de datos. La capacidad para crear estos nuevos tipos es una característica importante y potente de C++ y libera a un programador de restringirse al uso de los tipos ofrecidos por el lenguaje. Una *estructura* contiene múltiples variables, que pueden ser de tipos diferentes. La estructura es importante para la creación de programas potentes, tales como bases de datos u otras aplicaciones que requieran grandes cantidades de datos. Además, se estudiará el uso de la estructura como un tipo primitivo de dato denominado *objeto*. Por otra parte, se analizará el concepto de *unión*, otro tipo de dato no tan importante como las estructuras `array` y `struc`, pero sí necesarias en algunos casos.

Un tipo de dato enumerado es una colección de miembros con nombre que tienen valores enteros equivalentes. Un `typedef` es de hecho no un nuevo tipo de dato sino simplemente un sinónimo de un tipo existente.

CONCEPTOS CLAVE

- Estructura.
- Estructuras anidadas.
- `sizeof`.
- `struct`.
- `typedef`.
- Unión.
- `union`.
- Visibilidad.

8.1. ESTRUCTURAS

Los arrays son estructuras de datos que contienen un número determinado de elementos (su tamaño) y todos los elementos han de ser del mismo tipo de datos. Esta característica supone una gran limitación cuando se requieren grupos de elementos con tipos diferentes de datos cada uno. Por ejemplo, si se dispone de una lista de temperaturas, es muy útil un array; sin embargo, si se necesita una lista de información de clientes que contengan elementos tales como el nombre, la edad, la dirección, el número de la cuenta, etc., los arrays no son adecuados. La solución a este problema es utilizar un tipo de dato *estructura*.

Los componentes individuales de una estructura se llaman *miembros*. Cada miembro (elemento) de una estructura puede contener datos de un tipo diferente de otros miembros. Por ejemplo, se puede utilizar una estructura para almacenar diferentes tipos de información sobre una persona, tal como nombre, estado civil, edad y fecha de nacimiento. Cada uno de estos elementos se denominan *nombre del miembro*.

> Una **estructura** es una colección de uno o más tipos de elementos denominados **miembros**, cada uno de los cuales puede ser un tipo de dato diferente.

Una estructura puede contener cualquier número de miembros, cada uno de los cuales tiene un nombre único, denominado *nombre* del miembro. Supongamos que se desea almacenar los datos de una colección de discos compactos (**CD**) de música. Estos datos pueden ser:

- título,
- artista,
- número de canciones,
- precio,
- fecha de compra.

La estructura CD contiene cinco miembros. Tras decidir los miembros, se debe decidir cuáles son los tipos de datos para utilizar por los miembros. Esta información se representa en la tabla siguiente:

Nombre miembro	**Tipo de dato**
Título	Array de caracteres de tamaño 30.
Artista	Array de caracteres de tamaño 25.
Número de canciones	Entero.
Precio	Coma flotante.
Fecha de compra	Array de caracteres de tamaño 8.

La Figura 8.1 contiene la estructura `CD`, mostrando gráficamente los tipos de datos dentro de la estructura. Obsérvese que cada miembro es un tipo de dato diferente.

Título	`Ay, ay, ay, se me ha muerto el canario`
Artista	`No me pises que llevo chanclas`
Número de canciones	`10`
Precio	`2222.25`
Fecha de compra	`8-10-1992`

Figura 8.1. Representación gráfica de una estructura CD.

8.1.1. Declaración de una estructura

Una estructura es un tipo de dato definido por el usuario, que se debe declarar antes de que se pueda utilizar. El formato de la declaración es:

```
struct <nombre de la estructura>
{
   <tipo de dato miembro1> <nombre miembro1>;
   <tipo de dato miembro2> <nombre miembro2>;
   ...
   <tipo de dato miembron> <nombre miembron>;
};
```

La declaración de la estructura `CD` es

```
struct coleccion_CD
{
   char titulo[30];
   char artista[25];
   int num_canciones;
   float precio;
   char fecha_compra[8];
}
```

Ejemplo

```
struct complejo
{
   float parte_real, parte_imaginaria;
};
```

8.1.2. Definición de variables de estructuras

Al igual que a los tipos de datos enumerados, a una estructura se accede utilizando una variable o variables que se deben definir después de la declaración de la estructura. Del mismo modo que sucede en otras situaciones, en C++ existen dos conceptos similares a considerar, *declaración* y *definición*. La diferencia técnica es la siguiente. Una declaración especifica simplemente el nombre y el formato de la estructura de datos, pero no reserva almacenamiento en memoria. Una definición de variable para una estructura dada crea un área en memoria en donde los datos se almacenan de acuerdo al formato estructurado declarado. Las variables de estructuras se pueden definir de dos formas: *(1)* listándolas inmediatamente después de la llave de cierre de la declaración de la estructura, o *(2)* listando el nombre de la estructura seguida por las variables correspondientes en cualquier lugar del programa antes de utilizarlas. La definición y declaración de la estructura `colecciones_CD` se puede hacer por cualquiera de los dos métodos:

```
1. struct colecciones_CD
      {
         char título[30];
         char artista[25];
         int num_canciones;
         float precio;
```

```
          char fecha_compra[8];
       } cd1, cd2, cd3;

2. colecciones_CD cd1, cd2, cd3;
```

Obsérvese que esta definición difiere del sistema empleado en el lenguaje C, en donde es obligatorio el uso de la palabra reservada `struct` en la definición.

```
struct colecciones_CD cd1, cd2, cd3;
```

Otro ejemplo de definición/declaración

Consideremos un programa que gestione libros y procese los siguientes datos: título del libro, nombre del autor, editorial y año de publicación. Una estructura `info_libro` podría ser:

```
struct info_libro
{
   char título[60];
   char autor[30];
   char editorial[30];
   int anyo;
};
```

La definición de la estructura se puede hacer así:

```
1. info_libro libro1, libro2, libro3;

2. struct info_libro {
      char título[60];
      char autor[30];
      char editorial[30];
      int anyo;
      libro1, libro2, libro3;
}
```

8.1.3. Uso de estructuras en asignaciones

Como una estructura es un tipo de dato similar a un `int` o un `char`, se puede asignar una estructura a otra. Por ejemplo, se puede hacer que `libro3`, `libro4` y `libro5` tengan los mismos valores en sus miembros que `libro1`. Por consiguiente, sería necesario realizar las siguientes sentencias:

```
libro3 = libro1;
libro4 = libro1;
libro5 = libro1;
```

De modo alternativo se puede escribir

```
libro4 = libro5 = libro6 = libro1;
```

8.1.4. Inicialización de una declaración de estructuras

Se puede inicializar una estructura de dos formas. Se puede inicializar una estructura dentro de la sección de código de su programa, o bien se puede inicializar la estructura como parte de la declaración. Cuan-

do se inicializa una estructura como parte de la declaración, se especifican los valores iniciales, entre llaves, después de la declaración. El formato general en este caso:

```
struct <nombre variable estructura>
{  valor miembro1,
   valor miembro2,
   ...
   valor miembron };
```

Ejemplo 8.1

Inicializar una estructura en el momento que se declara.

```
Struct Fecha
{
   int mes;
   int día;
   int anyo;
};
```

Una vez que el tipo `Fecha` se ha definido, se puede declarar e inicializar una variable estructura llamada `FechaNacimiento` como sigue:

```
Fecha FechaNacimiento = {8,10,1956};
```

Los valores que inicializan se deben dar en el orden que corresponde al orden de las variables miembro de la definición del tipo estructura. En el ejemplo anterior, `FechaNacimiento.mes` toma el valor inicial de `8`, `FechaNacimiento.día` toma el valor `10` y `FechaNacimiento.anyo` toma el valor `1956`.

Se produce un error si hay más valores de inicialización que miembros de la estructura. Si hay menos valores de inicialización que miembros de la estructura, los valores proporcionados se utilizan para inicializar miembros datos en el orden señalado. Cada miembro dato sin un inicializador se inicializa a un valor cero del tipo apropiado de la variable.

```
struct info_libro
{
   char título[60];
   char auto[30];
   char editorial[30];
   int anyo;
} libro1 = { "C++ a su alcance", "Luis Joyanes","McGraw-Hill", 1994 };
```

Otro ejemplo podría ser:

```
struct coleccion_CD
{
   char titulo[30];
   char artista[25];
   int num_canciones;
   float precio;
   char fecha_compra[8];
```

```
} cd1 =  {
            "El humo ciega tus ojos",
            "Col Porter",
            15,
            2545,
            "02/6/94"
         };
```

8.1.5. El tamaño de una estructura

El siguiente programa ilustra el uso del operador `sizeof` para determinar el tamaño de una estructura:

```
// sizeof.cpp
#include <iostream>
using namespace std;

//declarar una estructura Persona
struct Persona
{
   char nombre[30];
   int edad;
   float altura;
   float peso;
};

void main()
{
   Persona persona;
   cout << "sizeof(Persona): " << sizeof(Persona);
}
```

Al ejecutar el programa se produce la salida:

```
sizeof(Persona) : 40
```

El resultado se obtiene determinando el número de bytes que ocupa la estructura

Persona	**Miembros dato**	**Tamaño (bytes)**
`nombre[30]`	`char(1)`	30
`edad`	`int(2)`	2
`altura`	`float(4)`	4
`peso`	`float(4)`	4
Total		40

8.2. ACCESO A ESTRUCTURAS

Cuando se accede a una estructura, o bien se almacena información en la estructura o se recupera la información de la estructura. Se puede acceder a los miembros de una estructura de una de estas dos formas: *(1)* utilizando el operador punto (.), o bien *(2)* utilizando el operador puntero `->`.

8.2.1. Almacenamiento de información en estructuras

Se puede almacenar información en una estructura mediante inicialización, asignación directa o lectura del teclado. El proceso de inicialización ya se ha examinado, veamos ahora la asignación directa y la lectura del teclado.

Acceso a una estructura de datos mediante el operador punto

La asignación de datos a estructuras se puede hacer mediante el operador punto. El operador (`.`) se utiliza para especificar una variable miembro de una variante estructura la sintaxis en C++ es:

```
<nombre variable estructura> . <nombre miembro> = datos;
                             ↑
                             └── operador punto
```

Algunos ejemplos son:

```
strcpy (cd1.titulo, "Granada");[1]
cd1.precio = 3450.75;
cd1.num_canciones = 7;
```

El operador punto proporciona el camino directo al miembro correspondiente. Los datos que se almacenan en un miembro dado deben ser del mismo tipo que el tipo declarado para ese miembro.

Ejemplo 8.2

```
struct RegEstudiante
{
   int NumExpEstudiante;
   char curso;
};

int main()
{  RegEstudiante RegPersonal;
   RegPersonal.NumExpEstudiante = 2010;
}  RegPersonal.curso = 'Doctorado';
```

Acceso a una estructura de datos mediante el operador puntero

El operador puntero, `->`, sirve para acceder a los datos de la estructura. Para utilizar este operador se debe definir primero una variable puntero para apuntar a la estructura. A continuación, utilice simplemente el operador puntero para apuntar a un miembro dado. La asignación de datos a estructuras utilizando el operador puntero tiene el formato:

```
<puntero estructura> -> <nombre miembro> = datos;
```

[1] Al declarar `titulo` como un array de caracteres, la asignación de la cadena “Granada” a `titulo` sólo se puede hacer mediante la función `strcpy`. Si `titulo` se hubiera declarado de tipo `string`, entonces se podría haber realizado la asignación directa `cd1.titulo = "Granada"`. En el Capítulo 10 se estudiarán en profundidad las cadenas.

Así, por ejemplo, una estructura `estudiante`

```
struct estudiante
{
   char *Nombre;
   int Num_Estudiante;
   int Anyo_de_matricula;
   float Nota;
};
```

Se puede definir `Mortimer` como un puntero a la estructura

```
Estudiante *Mortimer;
```

A los miembros de la estructura `Estudiante` se pueden asignar datos como sigue (siempre y cuando la estructura ya tenga creado su espacio de almacenamiento, por ejemplo, con `malloc()`):

```
Mortimer -> Num_Estudiante = 3425;
Mortimer -> Nota = 7.5;
strcpy(Mortimer -> Nombre, "Pepe Mortimer");
```

> **Nota**
>
> Previamente habría que crear espacio de almacenamiento en memoria; por ejemplo, con la función `malloc()`.

Ejemplos de declaración de estructuras

```
struct Automovil                    Struct Automovil
{                                   {
   int anyo;
   int puerta,        equivale a       int anyo, puertas;
   double PotenciaCV;                  double PotenciaCV;
   char modelo;                        char modelo;
};                                  {
```

```
Variable de tipo estructura
Automovil miCarro, tuCarro, Su Carro;
```

Las variables miembro se especifican con el operador punto

```
miCarro.anyo
miCarro.PotenciaCV
miCarro.modelo
```

Otro tipo de declaración de variables estructuras:

```
Struct DatosLluvia
{
   double temperatura;
   double velocidadDelViento;
} hora1, hora2;
```

8.2.2. Lectura de información de una estructura

Si ahora se desea introducir la información en la estructura mediante el teclado, basta con emplear una sentencia de entrada utilizando el operador punto o puntero. Así, si `Mortimer` es una variable ordinaria de tipo estructura

```
Estudiante Mortimer;
...
cout << "\n Introduzca el nombre del estudiante: ";
gets(Mortimer.Nombre);
cout << "\n Introduzca el número de estudiante: ";
cin >> Mortimer.Num_Estudiante;
...
```

Ahora, si `Mortimer` está definido como una variable puntero a la estructura

```
Estudiante *Mortimer;
cout << "\n Introduzca el nombre del estudiante: ";
gets(Mortimer -> Nombre);
cout << "\n Introduzca el número de estudiante: ";
cin >> Mortimer -> Num_Estudiante;
...
```

8.2.3. Recuperación de información de una estructura

Se recupera información de una estructura utilizando el operador de asignación o una sentencia `cout`. Igual que antes, se puede emplear el operador punto o el operador puntero. El formato general toma uno de estos formatos:

```
1. <nombre variable> =
           <nombre variable estructura>.<nombre miembro>;
```

o bien

```
   <nombre variable> =
         <puntero de estructura> -> <nombre miembro>;
```

```
2. cout << <nombre variable estructura>.<nombre miembro>
```

o bien

```
   cout << <puntero de estructura> -> <nombre miembro>;
```

Algunos ejemplos del uso de la estructura `Estudiante` son:

```
Número = Mortimer.Num_Estudiante;
Grado = Mortimer -> Nota;
cout << Mortimer.Nombre;
cout << Mortimer -> Anyo_de_matricula;
```

8.3. ESTRUCTURAS ANIDADAS

Una estructura puede contener otras estructuras llamadas *estructuras anidadas*. Las estructuras anidadas ahorran tiempo en la escritura de programas que utilizan estructuras similares. Se han de definir los miembros comunes sólo una vez en su propia estructura y, a continuación, utilizar esa estructura como un miembro de otra estructura.

Consideremos las siguientes dos definiciones de estructuras:

```
struct empleado
{
   char nombre_emp[30];          //Nombre del empleado
   char dirección[25];           //Dirección del empleado
   char ciudad[20];
   char provincia[20];
   long int cod_postal;
   double salario;               //Salario anual
};

struct clientes
{
   char nombre_cliente[30];      //Nombre del cliente
   char direccion[25];           //Dirección del cliente
   char ciudad[20];
   char provincia[20];
   long int cod_postal;
   double saldo;                 //Saldo en la compañía
};
```

Estas estructuras contienen muchos datos diferentes, aunque hay datos que están solapados. Así, se podría disponer de una estructura, `info_dir`, que contuviera los miembros comunes.

```
struct info_dir
{
   char direccion[25];
   char ciudad[20];
   char provincia[20];
   long int cod_postal;
};
```

Esta estructura se puede utilizar como un miembro de las otras estructuras, es decir, *anidarse*.

```
struct empleado
{
   char nombre_emp[30];
   struct info_dir direccion_emp;
   double salario;
};

struct clientes
{
   char nombre_cliente[30];
   struct info_dir direccion_clien;
   double saldo;
};
```

Gráficamente se podrían mostrar estructuras anidadas en la Figura 8.2.

```
empleado:                              cliente:
   nombre_emp                             nombre_cliente
             | dirección                            | dirección
   info_dir  | ciudad                      info_dir | ciudad
             | provincia                            | provincia
             | cod_postal                           | cod_postal
   salario                                saldo
```

Figura 8.2. Estructuras anidadas.

8.3.1. Un ejemplo de estructuras anidadas

Se desea diseñar una estructura que contenga información de operaciones financieras. Esta estructura debe constar de un número de cuenta, una cantidad de dinero, el tipo de operación (depósito o retirada de fondos) y la fecha y hora en que la operación se ha realizado. A fin de realizar el acceso correcto a los campos `dia`, `mes` y `anyo`, así como `tiempo` (la hora y minutos) en que se efectuó la operación, se define una estructura `registro_operacion` de la forma siguiente:

```
#include <iostream>
using namespace std;

enum tipo_de_operacion {DEPOSITO, RETIRADA};
struct registro_operacion {
   long número_cuenta;
   float cantidad;
   tipo_de_operacion tipo_operacion;
   struct {
      int mes, dia, anyo;
   } fecha;
   struct {
      int horas, minutos;
   } tiempo;
};

void main() {
   struct registro_operacion operacion;
   operacion.cantidad = 500.00;
   operacion.tipo_transaccion = DEPOSITO;
   operacion.fecha.mes = 10;
   operacion.fecha.dia = 25;
   operacion.tiempo.horas = 8;
   operacion.tiempo.minutos = 45;
}
```

Ejercicio 8.1

Se desea registrar una estructura `PersonaEmpleado` *que contenga como miembros los datos de una persona empleado que a su vez contenga los datos de la fecha de nacimiento.*

```
Persona Empleado

    Persona

    Fecha
```

```
struct Fecha
{
   unsigned int dia;                  // enteros positivos
   unsigned int mes;
   unsigned int anyo;
};

struct Persona {
   char     nombre[20];               // entero positivo
   unsigned int edad;
   float    altura;
   float    peso;
   Fecha    fec;
};

struct PersonaEmpleado
{
   Persona      persona;
   unsigned int salario;
   unsigned int horas_por_semana;
};
```

El siguiente programa muestra el uso de estas estructuras:

```
// estruc.cpp
// uso de estructuras anidadas
#include <iostream>
using namespace std;

struct Fecha
{
   unsigned int dia;               // enteros positivos
   unsigned int mes;
unsigned int anyo;
};

struct Persona {
   char     nombre[20];            // entero positivo
   unsigned int edad;
   float    altura;
   float    peso;
   Fecha    fec;
};
```

```
struct PersonaEmpleado
{
   Persona      persona;
   unsigned int salario;
   unsigned int horas_por_semana;
};

void main()
{

// definir una variable PersonaEmpleado

PersonaEmpleado pe;

// establecer miembros
cout << "introduzca su nombre: "; cin >> pe.persona.nombre;
cout << "introduzca su edad: ";    cin >> pe.persona.edad;
cout << "introduzca su altura: "; cin >> pe.persona.altura;
cout << "introduzca su peso: ";    cin >> pe.persona.peso;
cout << "introduzca su fecha de nacimiento: ";
cin  >> pe.persona.fec.dia;
cin  >> pe.persona.fec.mes;
cin  >> pe.persona.fec.anyo;
cout << "introduzca su salario: ";
cin  >> pe.salario;
cout << "introduzca número de horas: ";
cin  >> pe.horas_por_semana;
// salida
cout << endl;
cout << "nombre: " << pe.persona.nombre << endl;
cout << "edad:  " << pe.persona.edad << endl;
cout ...
cout << "fecha de nacimiento: "
cout << pe.persona.fec.dia << "-";
     ..
}
```

El acceso a miembros dato de estructuras anidadas requiere el uso de múltiples operadores punto. *Ejemplo*: acceso al día del mes de la fecha de nacimiento de un empleado.

```
pe.persona.fec.dia
```

Las estructuras se pueden anidar a cualquier grado. También es posible inicializar estructuras anidadas en la definición. El siguiente ejemplo inicializa una variable `Luis` de tipo `Persona`.

```
// ...
Persona Luis { "Luis", 25, 1.940, 40, { 12', 1, 70}};
```

8.3.2. Estructuras jerárquicas

A veces es importante tener estructuras cuyos miembros son, por sí mismos, estructuras más pequeñas. Por ejemplo, una estructura tipo `InfoPersona` se puede utilizar para almacenar los datos personales de un estudiante, altura, peso y fecha de nacimiento.

```
struct Fecha
{
   int mes;
   int día;
   int anyo;
};

struct InforPersona
{
   double altura;   //altura en metros
   int peso;        //peso en kilos
   Fecha fechaDeNacimiento;
};
```

Una variable estructura de tipo `InfoPersona` se puede declarar del modo usual como:

```
InfoPersona persona1;
```

Ejemplo

Visualizar el año de nacimiento de una palabra

```
cont << persona1.fechaDeNacimiento.anyo
```

variable miembro de nombre `fechaDeNacimiento`

8.4. ARRAYS DE ESTRUCTURAS

Se puede crear un array de estructuras tal como se crea un array de otros tipos. Los arrays de estructuras son idóneos para almacenar un archivo completo de empleados, un archivo de inventario, o cualquier otro conjunto de datos que se adapte a un formato de estructura. Mientras que los arrays proporcionan un medio práctico de almacenar diversos valores del mismo tipo, los arrays de estructuras le permiten almacenar juntos diversos valores de diferentes tipos, agrupados como estructuras.

Muchos programadores de C++ utilizan arrays de estructuras como un método para almacenar datos en un archivo de disco. Se pueden introducir y calcular sus datos de disco en arrays de estructuras y, a continuación, almacenar esas estructuras en memoria. Los arrays de estructuras proporcionan también un medio de guardar datos que se leen del disco.

La declaración de un array de estructuras `info_libro` se puede hacer de un modo similar a cualquier array, es decir,

```
info_libro libros[100];
```

asigna un array de 100 elementos denominado `libros`. Para acceder a los miembros de cada uno de los elementos estructura se utiliza una notación de array. Para inicializar el primer elemento de `libros`, por ejemplo, su código debe hacer referencia a los miembros de `libros[0]` de la forma siguiente:

```
strcpy(libros[0].titulo, "C++ a su alcance");
strcpy(libros[0].autor, "Luis Joyanes");
strcpy(libros[0].editorial, "McGraw-Hill");
libros[0].anyo = 1994;
```

Se puede también inicializar un array de estructuras en el punto de la declaración encerrando la lista de inicializadores entre llaves, { }. Por ejemplo,

```
info_libro libros[3] = { "C++ a su alcance", "Luis Joyanes",
      "McGraw-Hill", 1994, "The Annotated C++. Reference
      Manual", "Stroustrup", "Addison-Wesley", 1992, "The
      Design and Evolution of C++", "Stroustrup", "Addison-
      Wesley", 1994, "Object Orientation", "Hares, Smart",
      "Wiley", 1994};
```

8.4.1. Arrays como miembros

Los miembros de las estructuras puede ser asimismo arrays. En este caso, será preciso extremar las precauciones cuando se accede a los elementos individuales del array.

Considérese la siguiente definición de estructura. Esta sentencia declara un array de 100 estructuras, cada estructura contiene información de datos de empleados de una compañía.

```
struct nomina
{
   char nombre[30];               //Array nombre
   int dependientes;
   char departamento[10];         //Array departamento
   float salario;
} empleado[100];                  //Un array de 100 empleados
```

Ejemplo 8.3

Una librería desea catalogar su inventario de libros. El siguiente programa crea un array de 100 estructuras, donde cada estructura contiene diversos tipos de variables, incluyendo arrays.

```
// Nombre del archivo: LIBROS.CPP

#include <iostream>
#include <stdio.h>
#include <ctype.h>
using namespace std;

struct inventario
{
   char titulo[25];
   char fecha_pub[20];
   char autor[30];
   int num;
   int pedido;
   float precio_venta;
};

main()
{
   struct inventario libro[100];
   int total = 0;
   char resp;
```

```
   do {
      cout << "Total libros " << (total+1) << "\n";
      cout << "¿Cuál es el título? ";
      gets(libro[total].titulo);

      cout << "¿Cuál es la fecha de publicación? ";
      gets(libro[total].fecha_pub);

      cout << "¿Quién es el autor? ";
      gets(libro[total].autor);

      cout << "¿Cuántos libros existen? ";
      cin >> libro[total].num;
      cout << "¿Cuántos ejemplares existen pedidos? ";
      cin >> libro[total].pedido;

      cout << "¿Cuál es el precio de venta? ";
      cin >> libro[total].precio_venta;
      fflush(stdin);

      cout << "\n ¿Hay más libros? (S/N)";
      cin >> resp;
      fflush(stdin);
      resp = toupper(resp);
      if (resp == 'S')
      {
         total++;
         continue;
      }
   } while (resp == 'S');
   return 0;
}
```

8.5. UTILIZACIÓN DE ESTRUCTURAS COMO PARÁMETROS

C++ permite pasar estructuras bien por valor o bien por referencia. Si la estructura es grande, el tiempo necesario para copiar un parámetro `struct` a la pila puede ser prohibitivo. En tales casos, se debe considerar el método de paso por referencia.

El listado `ESTRUCT1.CPP` muestra un programa que pasa una estructura a una función utilizando el método de paso por valor.

```
// ESTRUCT1.CPP
// Muestra el paso por valor de una estructura
#include <iostream>
using namespace std;

// Definir el tipo estructura InfoPersona
struct InfoPersona {
   char nombre[20];
   char calle[30];
   char ciudad[25];
```

```
      char provincia[25];
      char codigoPostal[5];
   };

   void verInfo(InfoPersona Datos)
   {
      cout << Datos.nombre << endl
             << Datos.calle << endl
             << Datos.ciudad << " " << Datos.provincia
             << " " << Datos.codigoPostal << endl;
   }

   void main(void)
   {
      //Definir RegistroDatos como tipo estructura
      InfoPersona RegistroDatos = {"Pepe Luis MacKoy",
                                   "3 de mayo","Cazorla",
                                   "Jaén", "63441"};
      VerInfo(InfoPersona);

      cout << "Pulsa Intro(Enter) para continuar";
      cin.get();
   }
```

Si desea pasar la estructura por referencia, necesita situar un operador de referencia & entre `InfoPersona` y `Datos` en la cabecera de la función `VerInfo()`.

8.6. VISIBILIDAD DE UNA ESTRUCTURA: ¿PÚBLICA O PRIVADA?

En C++, las estructuras se utilizan normalmente para contener tipos de datos diferentes, mientras que las clases se utilizan para contener datos y funciones. De hecho, C++ permite también estructuras que tienen miembros función y dato.

Ejemplo 8.4

Diseñar una estructura `Punto` (coordenadas en 3 dimensiones x, y, z) que proporcione dos funciones miembro `Sumar()` y `Restar()` que suman y restan, respectivamente, dos objetos de tipo `Punto`:

```
   // declaración de una estructura con miembros dato y función
   #include <iostream>
   using namespace std;

   struct Punto
   {
      // miembros dato
      double x, y, z;

      // funciones miembro
      void sumar (const Punto& p1, const Punto& p2)
      {
         x = p1.x + p2.x;
```

```
      y = p1.y + p2.y;
      z = p1.z + p2.z;
   }

   void Restar (const Punto& p1, const Punto& p2)
   {
      x = p1.x – p2.x;
      y = p1.y – p2.y;
      z = p1.z – p2.z;
   }
};

void main()
{
   Punto p1, p2, p3;

   p1.x = 1.0; p1.y = 2.0; p1.z = 3.0;
   p2.x = 8.0; p2.y = 9.0; p2.z = 10.0;
   p3.x = 0.0; p3.y = 0.0; p3.z = 0.0;

   cout << "antes: " << endl;
   cout << "p3.x: " << p3.x << endl;
   cout << "p3.y: " << p3.y << endl;
   cout << "p3.z: " << p3.z << endl;
   p3.sumar (p1, p2);

   cout << "despues: " << endl;
   cout << "p3.x: " << p3.x << endl;
   cout << "p3.y: " << p3.y << endl;
   cout << "p3.z: " << p3.< << endl;
}
```

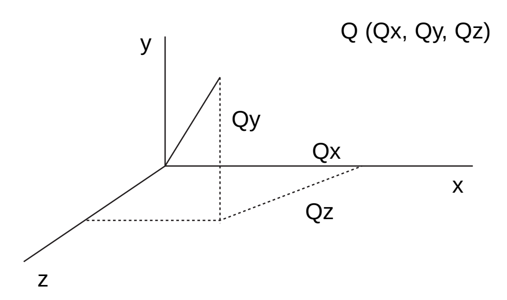


Figura 8.3. Punto Q en tres dimensiones.

La declaración del tipo `Punto` incluye dos funciones miembro `Sumar()` y `Restar()`. A estas funciones se pasan dos argumentos de tipo `Punto` y manipulan estos dos puntos añadiendo o restando sus respectivos miembros dato `(x, y, z)`.

En C++, las estructuras se reservan generalmente un mecanismo de encapsulación de datos que contiene sólo funciones miembro, mientras que la clase (`class`) se utiliza para definir tipos que contienen ambos datos y funciones.

Privado o público

Si una estructura se declara con ámbito global, sus miembros dato son accesibles desde cualquier función. En otras palabras, sus miembros son *públicos*. Los miembros de una estructura son por defecto públicos y son accesibles, por consiguiente, a un objeto. En el ejemplo de la sección anterior, se añadieron dos funciones miembro a la estructura `Punto` para sumar y restar sus miembros dato. Cuando las funciones y datos miembro se declaran públicos por defecto, se pueden manipular desde el exterior los miembros dato desde el exterior de la declaración de la estructura. Así, un ejemplo:

```
// ...
// definir e inicializar dos puntos
p1.x = 1.0; p1.y = 2.0; p1.z = 3.0;
p2.x = 8.0; p2.y = 9.0; p2.z = 10.0;
p3.x = 0.0; p3.y = 0.0; p3.z = 0.0;
// ...
Punto p3;
p3.x = p1.x + p2.x;
p3.y = p1.y + p2.y;
p3.z = p1.z + p2.z;
// o bien
p3.Sumar (p1,p2);
```

8.7. UNIONES

Las uniones son casi idénticas a las estructuras, pero producen resultados diferentes. Una estructura (`struct`) permite almacenar variables relacionadas juntas y almacenadas en posiciones contiguas en memoria. Las uniones declaradas con la palabra reservada `union` almacenan también miembros múltiples en un paquete; sin embargo, en lugar de situar sus miembros unos detrás de otros, en una unión, todos los miembros se solapan entre sí en la misma posición.

La sintaxis de una unión es

```
union nombre {
   tipo1 miembro1;
   tipo2 miembro2;
   ...
};
```

Un ejemplo es

```
union PruebaUnion {
   float Item1;
   long Item2;
};
```

La cantidad de memoria reservada para una unión es igual a la anchura de la variable más grande. En el tipo `union`, cada uno de los miembros dato comparten memoria con los otros miembros de la unión. La cantidad total de memoria utilizada por la unión `PruebaUnionTest` es de 4 bytes, ya que el elemento `float` es el miembro dato mayor de la unión.

```
union PruebaUnionTest {
   char letra;
```

```
   int elemento;
   float precio;
   char tecla;
};
```

Una razón para utilizar una unión es ahorrar memoria. En muchos programas se deben tener varias variables, pero no necesitan utilizarse todas al mismo tiempo. Consideremos la situación en que se necesitan tener diversas cadenas de caracteres de entrada. Se pueden crear varios arrays de cadenas de caracteres, tales como las siguientes:

```
char linea_ordenes[80];
char mensaje_error[80];
char ayuda[80];
```

Estas tres variables ocupan 240 bytes de memoria. Sin embargo, si su programa no necesita utilizar las tres variables simultáneamente, ¿por qué no permitirle compartir la memoria utilizando una unión? Cuando se combinan en el tipo union `frases`, estas variables ocupan un total de 80 bytes.

```
union frases {
   char linea_ordenes[80];
   char mensaje_error[80];
   char ayuda[80];
} cadenas;
```

Para referirse a los miembros de una unión, se utiliza el operador punto (.).

```
cadenas.ayuda
cadenas.mensaje_error
```

8.8. ENUMERACIONES

Una enumeración, **`enum`**, es un tipo definido por el usuario con constantes de nombre de tipo entero.

Formato

1.
```
enum {
      enumerador1, enumerador2, enumerador3,...
   };
```

2.
```
enum Nombre
   {
     enumerador1, enumerador2, enumeradorn,...
   };
```

3.
```
enum Nombre
   {
      enumerador1 = expresión_constante1,
      enumerador2 = expresión_constante2,
      ...
      enumeradorn = exprsesión_constanten
   };
```

Ejemplo 8.5

***Usos típicos de* `enum`**

```
enum Interruptor
{
   ENCENDIDO;      // ON, "prendido"
   APAGADO;        // OFF, "apagado"
};

enum Boolean
{
   FALSE;
   TRUE;
};
```

Ejemplo

```
enum {
   ROJO, VERDE, AZUL;
};
```

define tres constantes —`ROJO`,`VERDE` y `AZUL`— de valores iguales a 0, 1 y 2, respectivamente. Los miembros datos de un `enum` se llaman enumeradores y la constante entera por defecto del primer enumerador de la lista de los miembros datos es igual a 0. Obsérvese que, al contrario que `struct` y `union`, los miembros de un tipo `enum` se separan por el operador coma. El ejemplo anterior es equivalente a la definición de las tres constantes, `ROJO`, `VERDE` y `AZUL`, tal como:

```
const int ROJO = 0;
const int VERDE = 1;
const int AZUL = 2;
```

Una enumeración puede tener nombre

```
enum DiasSemana
{
   LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
};
```

A los enumeradores se pueden asignar valores o expresiones constantes durante la declaración:

```
enum Hexaedro
{
   VERTICES = 8,       // número de vértices
   LADOS    = 12,      // número de lados
   CARAS    = 6        // número de caras
   }
```

Ejercicio 8.2

El siguiente programa muestra el uso de la enumeración `Boolean`.

```
// enum.cpp
// muestra de enum
#include <iostream.h>      // E/S de C++
#include <conio.h>         // función getche()

// enumeración boolean
enum Boolean
{
   FALSE, TRUE
};

// la función Vocal devuelve TRUE si se pasa una vocal a una
// función
// devuelve FALSE en caso contrario
// Boolean Vocal  (char c)
{
   switch (c)
   {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
         return TRUE;
      default
         return FALSE;
      }
   }

   // función principal
   void main()
   {
      char car = 'x';         // inicializar car
      do
      {
         cout << endl << "introduzca una letra consonante
                           para fin:";
         car = getche();
      } while (Vocal (car));
   }
```

Ejecución

```
introduzca una letra consonante para fin: a
introduzca una letra consonante para fin: i
introduzca una letra consonante para fin: w  // al introducir w, fin
```

8.9. OPERADOR `SIZEOF`

El siguiente programa extrae el tamaño de una estructura (`struc`) y una unión (`unión`) con miembros dato idéntico.

```
// declarar una unión TipoUnión
union TiposUnion
{
   char c;
   int i;
   float f;
   double d;
};

// declarar una estructura TipoEstructura
struct TiposEstructura
{
   char c;
   int i;
   float f;
   double d;
};
...
//
cout << "sizeof(TiposStructura): " << sizeof(TipoEstructura)
     << endl;
cout << "sizeof(TiposUnion): " << sizeof(TiposUnion) << endl;
```

Estos argumentos de programa producirán la siguiente salida:

```
Sizeof(TiposEstructura) : 15
Sizeof(TiposUnion)      : 8
```

8.9.1. Sinónimo de un tipo de datos: `typedef`

Un `typedef` permite a un programador crear un sinómimo de un tipo de dato definido por el usuario o integral ya existente.

Ejemplo

Uso de `typedef` para declarar un nuevo nombre, longitud de tipo dato integral.

```
// ...
typedef double Longitud;
// ...
Longitud Distancia (const Punto& p, const Punto& p2)
{
   // ...
   Longitud longitud = sqrt(r-cua);
   return longitud;
}
```

Otros ejemplos son:

```
typedef char* String;               // puntero a char
typedef const char* String;         // puntero a const char
```

La desventaja de `typedef` es que introduce nombres de tipos adicionales y pueden hacer los tipos que existen más abstractos.

RESUMEN

Una estructura permite que los miembros dato de los mismos o diferentes tipos se encapsulen en una única implementación, al contrario que los arrays que son agregados de un tipo de dato simple. Los miembros dato de una estructura son, por defecto, accesibles públicamente, aunque una estructura puede tener diferentes tipos de acceso o visibilidad, distintos a `public`. Además, una estructura puede, de hecho, encapsular funciones que operan sobre los datos de la estructura, aunque estas propiedades se suelen reservar generalmente a `class`.

- Una estructura es un tipo de dato que contiene elementos de tipos diferentes.

```
struct empleado {
       char nombre[30];
       long salario;
       char num_telefono[10];
    };
```

- Para crear una variable estructura se escribe

```
struct empleado pepe;  // estilo C.
```

o bien

```
empleado pepe;         // estilo C++
```

- Para determinar el tamaño en bytes de cualquier tipo de dato C++ utilizar la sentencia `sizeof()`.
- El tipo de dato `union` es similar a **`struct`**, de modo que es una colección de miembros dato de tipos diferentes o similares, pero al contrario que una definición **`struct`**, que asigna memoria suficiente para contener todos los miembros dato, una **`union`** puede contener sólo un miembro dato en cualquier momento dado y el tamaño de una unión es, por consiguiente, el tamaño de su número mayor.
- Utilizando un tipo `union` se puede hacer que diferentes variables coexistan en el mismo espacio de memoria. La unión permite reducir espacio de memoria en sus programas.
- Un `typedef` no es nuevo tipo de dato sino un sinónimo de un tipo existente.

EJERCICIOS

8.1. Considere la siguiente definición:

```
struct TipoZapato
{
    char modelo;
    double precio;
};
```

¿Cuál será la salida producida por el siguiente código?

```
TipoZapato zapato1, zapato2-;
zapato1.estilo-= 'A';
zapato1.precio = 3450;
cout << zapato1.estilo << " pta "
     << zapato1.precio << endl-;
zapato2 = zapato1-;
cout << zapato2.estilo << " pta "
     << zapato2.precio << endl;
```

8.2. Dada una estructura estudiante como se define a continuación, y una instancia de la estructura, `un_estudiante`, escribir una sentencia que visualice el nombre del estudiante en el formato *apellido, nombre.*

```
struct estudiante
{
   string nombre;
   string apellido;
   int edad;
   int nota;
   char grado;
}-;  // fin de estudiante

estudiante   un_estudiante;
```

8.3. Escribir una función que visualice una variable de tipo `estudiante`.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 176).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

8.1. *Declarar un tipo de datos para representar las estaciones del año.*

8.2. *Escribir un función que devuelva la estación del año que se ha leído del teclado. La función debe de ser del tipo declarado en el Ejercicio 8.1.*

8.3. *Escribir una función que reciba el tipo enumerado del Ejercicio 8.1 y lo visualice.*

8.4. *Declarar un tipo de dato enumerado para representar los meses del año, el mes enero debe de estar asociado al dato entero 1, y así sucesivamente los demás meses.*

8.5. *Encuentre los errores del siguiente código:*

```
#include <stdio.h>
void escribe(struct fecha f);
int main()
{
  struct fecha
  {
   int dia;
   int mes;
   int anyo;
   char mes[];
   } ff;
   ff = {1,1,2000,"ENERO"};
   escribe(ff);
   return 1;
}
```

8.6. *¿Con* `typedef` *se declaran nuevos tipos de datos, o bien permite cambiar el nombre de tipos de datos ya declarados?*

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 177).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

8.1. *Escribir un programa que lea y escriba la información de 100 clientes de una determinada empresa. Los clientes tienen un nombre, el número de unidades solicitadas, el precio de cada unidad y el estado en que se encuentra: moroso, atrasado, pagado.*

8.2. *Añadir al programa anterior una función que permita mostrar la factura de todos los clientes de cada una de las categorías moroso, atrasado o pagado.*

8.3. *Modificar el programa de los Problemas 8.1 y 8.2 para obtener los siguientes listados:*

- *Clientes en estado moroso, con factura inferior a una cantidad.*
- *Clientes en estado pagado con factura mayor de una determinada cantidad.*

8.4. *Se desea registrar una estructura* `PersonaEmpleado` *que contenga como miembros los datos de una* `persona` *el salario y el número de horas*

trabajadas por semana. Una `persona`*, a su vez, es otra estructura que tiene como miembros el nombre, la edad, la altura, el peso, y la fecha de nacimiento. Por su parte, la fecha de nacimiento es otra estructura que contiene el día, el mes y el año. Escribir funciones para leer y escribir un empleado de tipo* `PersonaEmpleado`*.*

8.5. *Escribir funciones que permitan hacer las operaciones de suma, resta y multiplicación y cociente de números complejos en forma binómica, a+bi. El tipo complejo ha de definirse como una estructura.*

8.6. *Añadir al Problema 8.5 funciones para leer y escribir números complejos, y un programa que permita interactivamente realizar operaciones con números complejos.*

8.7. *Escribir un tipo de datos para representar números complejos en forma polar (módulo, argumento), y codificar dos funciones, para pasar un número complejo de forma polar a binómica y de forma binómica a polar respectivamente.*

8.8. *Escribir funciones que permitan hacer las operaciones de multiplicación y cociente y potencia de números complejos en forma polar.*

8.9. *Se quiere informatizar los resultados obtenidos por los equipos de baloncesto y de fútbol de la localidad alcarreña Lupiana. La información de cada equipo:*

- *Nombre del equipo.*
- *Número de victorias.*
- *Número de derrotas.*

Para los equipos de baloncesto añadir la información:

- *Número de perdidas de balón.*
- *Número de rebotes cogidos.*
- *Nombre del mejor anotador de triples.*
- *Número de triples del mejor triplista.*

Para los equipos de fútbol añadir la información:

- *Número de empates.*
- *Número de goles a favor.*
- *Número de goles en contra.*
- *Nombre del goleador del equipo.*
- *Número de goles del goleador.*

Escribir un programa para introducir la información para todos los equipos integrantes en ambas ligas.

8.10. *Modificar el programa del Problema 8.9 para obtener los siguientes informes o datos.*

- *Listado de los mejores triplistas de cada equipo.*
- *Máximo goleador de la liga de fútbol.*
- *Suponiendo que el partido ganado son tres puntos y el empate 1 punto: equipo ganador de la liga de fútbol.*
- *Equipo ganador de la liga de baloncesto.*

8.11. *Un punto en el plano se puede representar mediante una estructura con dos campos. Escribir un programa que realice las siguientes operaciones con puntos en el plano.*

- *Dados dos puntos calcular la distancia entre ellos.*
- *Dados dos puntos determinar el punto medio de la línea que los une.*

8.12. *Los protocolos IP de direccionamiento de red Internet definen la dirección de cada nodo de una red como un entero largo, pero dicho formato no es muy adecuado para los usuarios. Para la visualización a los usuarios se suelen separar los cuatro bytes separándolos por puntos. Escribir una función que visualice un entero largo como una dirección de red de los protocolos de Internet (cuatro bytes del entero largo separados por puntos).*

8.13. *Una librería desea tener el inventario de libros. Para ello quiere crear una base de datos en la que se almacenan la siguiente información por libro: título del libro; la fecha de publicación; el autor; el número total de libros existentes; el número total de libros existentes en pedidos; el precio de venta. Escribir funciones que permitan leer los datos de la base de datos y visualizar la base de datos.*

CAPÍTULO 9

Punteros (apuntadores)

Contenido

INTRODUCCIÓN

Los punteros en C y C++ tienen la fama, en el mundo de la programación, de dificultad, tanto en el aprendizaje como en su uso. En este capítulo se tratará de mostrar que los punteros no son más difíciles de aprender que cualquier otra técnica de programación ya examinada o por examinar a lo largo de este libro. El ***puntero*** o apuntador es una técnica muy potente que puede utilizar en sus programas para hacerlos más eficientes y flexibles. Los punteros son una de las razones fundamentales para que el lenguaje C/C++ sea tan potente y tan utilizado.

Una ***variable puntero*** (o ***puntero***, como se llama normalmente) es una variable que contiene direcciones de otras variables. Todas las variables vistas hasta este momento contienen valores de datos, por el contrario las variables punteros contienen valores que son direcciones de memoria donde se almacenan datos. En resumen, un puntero es una variable que contiene una dirección de memoria, y utilizando punteros su programa puede realizar muchas tareas que no sería posible utilizando tipos de datos estándar.

En este capítulo se estudiarán los diferentes aspectos de los punteros:

- concepto de puntero;
- utilización de punteros;
- asignación dinámica de memoria;
- aritmética de punteros;
- arrays de punteros;
- punteros a punteros, funciones y estructuras.

CONCEPTOS CLAVE

- Aritmética de punteros.
- Arrays de punteros.
- Direcciones.
- Palabra reservada `const`.
- Palabra reservada `null`.
- Palabra reservada `void`.
- Puntero (*apuntador*).
- Punteros *versus* arrays.
- Referencias.
- Tipos de punteros.

9.1. DIRECCIONES Y REFERENCIAS

Cuando una variable se declara, se asocian tres atributos fundamentales con la misma: su *nombre*, su *tipo* y su *dirección* en memoria.

Ejemplo

```
int n;      // asocia al nombre n, el tipo int y la dirección de
            // alguna posición de memoria donde se almacena el
            // valor de n
```

```
    0x4fffd34
  n [                  ]
    int
```

Esta caja representa la posición de almacenamiento en memoria. El nombre de la variable está a la izquierda de la caja, la dirección de variable está encima de la caja y el tipo de variable está debajo en la caja. Si el valor de la variable se conoce, se representa en el interior de la caja.

```
    0x4fffd34
  n [        75        ]
    int
```

Al valor de una variable se accede por medio de su nombre. Por ejemplo, se puede imprimir el valor de `n` con la sentencia

```
cout << n;
```

A la dirección de la variable se accede por medio del *operador de dirección* `&`. Por ejemplo, se puede imprimir la dirección de `n` con la sentencia

```
cout << &n;
```

El operador de dirección «&» «opera» (se aplica) sobre el nombre de la variable para obtener sus direcciones. Tiene precedencia de nivel 15 con el mismo nivel que el operador lógico `NOT (!)` y el operador de preincremento `++`. (*Véase* Apéndice C de la página Web del libro.)

Ejemplo 9.1

Obtener el valor y la dirección de una variable.

```
main()
{
   int n = 75;
   cout << "n = " << n << endl;        // visualiza el valor de n
   cout << "&n = " << &n << endl;      // visualiza dirección de n
}
```

Ejecución

```
n = 45
&n = 0x4fffd34
```

Nota

`0x4fffd34` es una dirección en código hexadecimal.
`“0x”` es el prefijo correspondiente al código hexadecimal.

9.1.1. Referencias

Una *referencia* es un alias de otra variable. Se declara utilizando el operador de referencia (`&`) que se añade al tipo de la referencia.

Ejemplo 9.2

Utilización de referencias.

```
void main()
{
   int n = 75;
   int& r = n;       // r es una referencia para n
   cout << "n = " << n << ", r = " << r << endl;
}
```

Ejecución

```
n = 75, r = 75
```

Nota

Los dos identificadores `n` y `r` son nombres diferentes para la misma variable.

Ejemplo 9.3

Las variables `n` *y* `r` *tienen la misma dirección de memoria*

```
void main()
{
   int n = 75;
   int& r = n;
   cout << "&n = " << &n << ", &r =" << &r << endl;
}
```

Ejecución

```
&n = 0x4fffd34, &r = 0x4fffd34
```

Regla

El carácter **&** tiene diferentes usos en C++: 1) cuando se utiliza como prefijo de un nombre de una variable, devuelve la dirección de esa variable; 2) cuando se utiliza como un sufijo de un tipo en una declaración de una variable, declara la variable como sinónimo de la variable que se ha inicializado; y 3) cuando se utiliza como sufijo de un tipo en una declaración de parámetros de una función, declara el parámetro referencia de la variable que se pasa a la función.

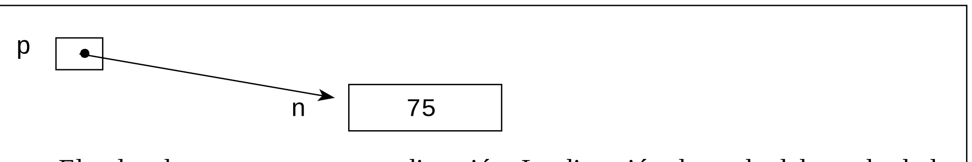
& *se refiere a la dirección en que se almacena el valor.*

9.2. CONCEPTO DE PUNTERO (APUNTADOR)[1]

Cada vez que se declara una variable C++, el compilador establece un área de memoria para almacenar el contenido de la variable. Cuando se declara una variable `int` (entera), por ejemplo, el compilador asigna dos bytes de memoria. El espacio para esa variable se sitúa en una posición específica de la memoria, conocida como *dirección de memoria*. Cuando se referencia (se hace uso) al valor de la variable, el compilador de C++ accede automáticamente a la dirección de memoria donde se almacena el entero. Se puede ganar en eficacia en el acceso a esta dirección de memoria utilizando un *puntero*.

Cada variable que se declara en C++ tiene una dirección asociada con ella. Un *puntero* es una dirección de memoria. El concepto de punteros tiene correspondencia en la vida diaria. Cuando se envía una carta por correo, su información se entrega basada en un puntero que es la dirección de esa carta. Cuando se telefonea a una persona, se utiliza un puntero (el número de teléfono que se marca). Así pues, una dirección de correos y un número de teléfono tienen en común que ambos indican dónde encontrar algo. Son punteros a edificios y teléfonos, respectivamente. Un puntero en C o en C++, también indica dónde encontrar algo, ¿dónde encontrar los datos que están asociados con una variable? *Un puntero C++ es la dirección de una variable*. Los punteros se rigen por estas reglas básicas:

- un *puntero* es una *variable* como cualquier otra;
- una variable puntero contiene una *dirección* que apunta a otra posición en memoria;
- en esa posición se almacenan los datos a los que apunta el puntero;
- un puntero apunta a una variable de memoria.



El valor de un puntero es una dirección. La dirección depende del estado de la computadora en la cual se ejecuta el programa.

El tipo de variable que almacena una dirección se denomina *puntero*

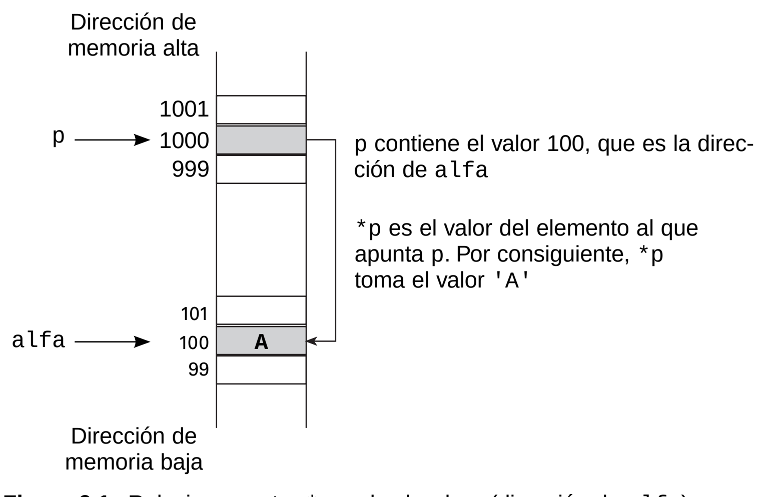


Figura 9.1. Relaciones entre `*p` y el valor de `p` (dirección de `alfa`).

[1] En Latinoamérica es usual emplear el término apuntador.

Ejemplo 9.4

```
void main()
{
   int n = 75;
   int* p = &n;     // p contiene la dirección de n
   cout << "n = " << n << ", &n = " << &n << ", p " << p <<
            endl;
   cout << "&p = " << &p << endl;
}
```

Ejecución

```
n = 75, &n = 0x4fffd34, p = 0x4fffd34
&p = 0x4fffd10
```

```
  0x4fffd30                      0x4fffd34
p | 0x4fffd34 |                n | 75 |
  int*                           int
```

La variable p se denomina *puntero* debido a que su valor «*apunta*» a la posición de otro valor. Es un puntero `int` cuando el valor al que apunta es de tipo `int` como en el ejemplo anterior.

9.2.1. Declaración de punteros

Al igual que cualquier variable, las variables punteros han de ser declaradas antes de utilizarlas. La declaración de una variable puntero debe indicar al compilador el tipo de dato al que apunta el puntero; para ello se hace preceder a su nombre con un asterisco (*), mediante el siguiente formato:

```
<tipo de dato apuntado> *<identificador de puntero>
```

Algunos ejemplos de variables punteros son:

```
int *ptr1;  // Puntero a un tipo de dato entero (int)
long *ptr2;  // Puntero a un tipo de dato entero largo (long int)
char *ptr3; // Puntero a un tipo de dato char
float *f;   // Puntero a un tipo de dato float
```

Un operador `*` (que está *sobrecargado*) en una declaración indica que la variable declarada almacena una dirección de un tipo de dato especificado. La variable *ptr1* almacena la dirección de un entero, la variable *ptr2* almacena la dirección, etc.

> Siempre que aparezca un asterisco (*) en una definición de una variable, ésta es una variable puntero.

9.2.2. Inicialización (iniciación) de punteros

Al igual que otras variables, C++ no inicializa los punteros cuando se declaran y es preciso inicializarlos antes de su uso. La *inicialización* de un puntero proporciona a ese puntero la dirección del dato corres-

pondiente. Después de la inicialización, se puede utilizar el puntero para referenciar los datos direccionados. Para asignar una dirección de memoria a un puntero se utiliza el operador de referencia &. Así, por ejemplo,

```
&valor
```

significa «la dirección de valor». Por consiguiente, el método de inicialización (iniciación), también denominado *estático*, requiere:

- Asignar memoria (estáticamente) definiendo una variable y, a continuación, hacer que el puntero apunte al valor de la variable.

```
int i;              //declara una variable i
int *p;             //declara un puntero a un entero p
p = &i;             //asigna la dirección de i a p
```

- Asignar un valor a la dirección de memoria.

```
*p = 50;
```

Cuando ya se ha definido un puntero, el asterisco delante de la variable puntero indica *«el contenido de»* de la memoria apuntada por el puntero y será del tipo dado.

Este tipo de inicialización es **estática**, ya que la asignación de memoria utilizada para almacenar el valor es fijo y no puede desaparecer. Una vez que la variable se define, el compilador establece suficiente memoria para almacenar un valor del tipo de dato dado. La memoria permanece reservada para esta variable y no se puede utilizar para otra cosa durante la ejecución del programa. En otras palabras, no se puede liberar la memoria reservada para una variable. El puntero a esa variable puede ser cambiado, pero la cantidad de memoria reservada permanecerá.

> El operador & devuelve la dirección de la variable a la cual se aplica.

Otros ejemplos de inicialización estática son:

```
1. int edad = 50;      //declara una variable edad de valor 50
   int *p_edad = &edad; //declara un puntero de enteros
                        //inicializándolo con la dirección de edad

2. char *p;            //Figura 9.1
   char alfa = 'A';
   p = &alfa;
```

Existe un segundo método para inicializar un puntero, es mediante *asignación dinámica de memoria*. Este método utiliza los operadores `new` y `delete`, y se analizará más adelante en este capítulo.

9.2.3. Indirección de punteros

Después de definir una variable puntero, el siguiente paso es inicializar el puntero y utilizarlo para direccionar algún dato específico en memoria. El uso de un puntero para obtener el valor al que apunta, es decir, su dato apuntado se denomina *indireccionar el puntero* («*desreferenciar* el puntero»); para ello se utiliza el operador de indirección `*` (también *sobrecargado*).

Las dos sentencias anteriores se describen en la Figura 9.2. Si se desea imprimir el valor de edad, se puede utilizar la siguiente sentencia `cout`:

```
cout << edad;          // imprime el valor de edad
```

También se puede imprimir el valor de edad con esta otra `cout`:

```
cout << *p_edad;       // indirecciona p_edad,
```

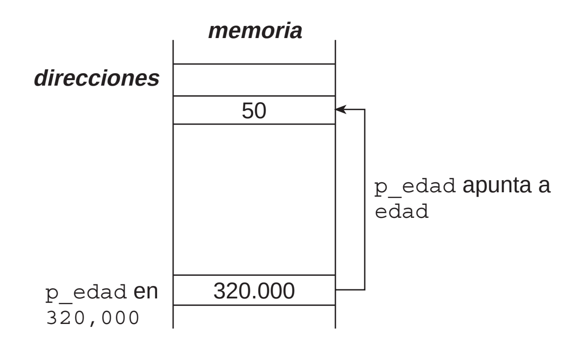


Figura 9.2. `p_edad` contiene la dirección de `edad`, `p_edad` apunta a la variable `edad`.

El listado `PUNTERO1.CPP` muestra el concepto de creación, inicialización e indirección de una variable puntero.

```
// PUNTERO1.CPP

#include <iostream.h>

char c;                  // una variable carácter

int main()
{
   char *pc;             // un puntero a una variable carácter

   pc = &c;
   for (c = 'A'; c <= 'Z'; c++);
       cout << *pc;
   return 0;
}
```

La ejecución de este programa visualiza el alfabeto. La variable puntero `pc` es un puntero a una variable carácter. La línea `pc = &c` asigna a `pc` la dirección de la variable `c` (`&c`). El bucle `for` almacena en `c` las letras del alfabeto y la sentencia `cout << *pc`; visualiza el contenido de la variable apuntada por `pc. c` y `pc` se refieren a la *misma* posición en memoria. Si la variable `c`, que se almacena en cualquier parte de la memoria, y `pc`, que apunta a esa misma posición, se refiere a los mismos datos, de modo que el cambio de una variable debe afectar a la otra. `pc` y `c` se dice que son *alias*, debido a que `pc` actúa como otro nombre de `c`.

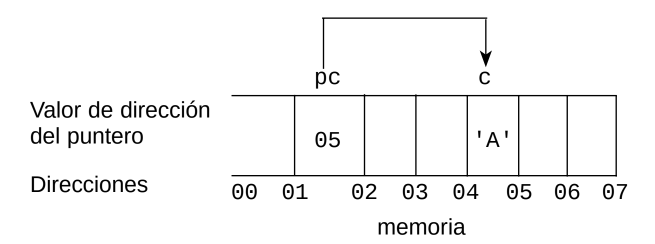


Figura 9.3. pc y c direccionan la misma posición de memoria.

La Tabla 9.1 resume los operadores de punteros

Tabla 9.1. Operadores de punteros.

Operador	Propósito
&	Obtiene la dirección de una variable.
*	Declara una variable como puntero.
*	Obtiene el contenido de una variable puntero.

> ***Nota***
>
> Son variables punteros aquellas que apuntan a la posición en donde otra/s variable/s de programa se almacenan.

9.2.4. Punteros y verificación de tipos

Los punteros se enlazan a tipos de datos específicos, de modo que C++ verificará si se asigna la dirección de un tipo de dato al tipo correcto de puntero. Así, por ejemplo, si se declara un puntero a `float`, no se le puede asignar la dirección de un carácter o un entero. Por ejemplo, este segmento de código no funcionará:

```
float *fp;
char c;
fp = &c;          // no es válido
```

C++ no permite la asignación de la dirección de `c` a `fp`, ya que `fp` es una variable puntero que apunta a datos de tipo real, `float`.

> C++ requiere que las variables puntero direccionen realmente variables del mismo tipo de dato que está ligado a los punteros en sus declaraciones.

9.3. PUNTEROS *NULL* Y *VOID*

Normalmente, un puntero inicializado adecuadamente apunta a alguna posición específica de la memoria. Sin embargo, un puntero no inicializado, como cualquier variable, tiene un valor aleatorio hasta que

se inicializa el puntero. En consecuencia, será preciso asegurarse que las variables puntero utilicen direcciones de memoria válida.

Existen dos tipos de punteros especiales muy utilizados en el tratamiento de sus programas: los punteros `void` y `null` (nulo).

Un *puntero nulo* no apunta a ninguna parte —objeto válido— en particular, es decir, «un puntero nulo no direcciona ningún dato válido en memoria». Un puntero nulo se utiliza para proporcionar a un programa un medio de conocer cuándo un puntero apunta a una dirección válida. Para declarar un puntero nulo se utiliza la macro `NULL`, definida en los archivos de cabecera `stdef.h`, `stdio.h`, `stdlib.h` y `string.h`. Se debe incluir uno o más de estos archivos de cabecera antes de que se pueda utilizar la macro `NULL`. Ahora bien, se puede definir `NULL` en la parte superior de su programa (o en un archivo de cabecera personal) con la línea

```
#define NULL 0
```

Un sistema de declarar un puntero nulo es:

```
char *p = NULL;
```

Algunas funciones C++ también devuelven el valor `NULL` si se encuentra un error. Se puede añadir una prueba o test comparando el puntero con `NULL`:

```
if (p == NULL) ...
```

o bien

```
if (p != NULL) ...
```

Otra forma de declarar un puntero nulo es asignar un valor de 0. Por ejemplo,

```
int *ptr = (int *) 0;      // ptr es un puntero nulo
```

Nunca se utiliza un puntero nulo para referenciar un valor. Como antes se ha comentado, los punteros nulos se utilizan en un test condicional para determinar si un puntero se ha inicializado. En el ejemplo

```
if (ptr)
   cout  << "Valor de la variable apuntada por ptr es: "
         << *ptr << "\n";
```

se imprime un valor si el puntero es válido y no es un puntero nulo.

Los punteros nulos se utilizan con frecuencia en programas con arrays de punteros.

En C++ se puede declarar un puntero de modo que apunte a cualquier tipo de dato, es decir, no se asigna a un tipo de dato específico. El método es declarar el puntero como un puntero `void *`.

```
void *ptr;         // declara un puntero void
```

El puntero `ptr` puede direccionar cualquier posición en memoria, pero el puntero no está unido a un tipo de dato específico. De modo similar, los punteros `void` pueden direccionar una variable `float`, una `char`, o una posición arbitraria o una cadena.

Nota

No confundir punteros `void` y `null`. Un puntero nulo no direcciona ningún dato válido. Un puntero `void` direcciona datos de un tipo no especificado. Un puntero `void` se puede igualar a nulo si no se direcciona ningún dato válido. Nulo es un valor; `void` es un tipo de dato.

9.4. PUNTERO A PUNTERO

Un puntero puede apuntar a otra variable puntero. Este concepto se utiliza con mucha frecuencia en programas complejos de C++. Para declarar un puntero a un puntero se hace preceder a la variable con dos asteriscos (**).

En el ejemplo siguiente `ptr5` es un puntero a un puntero.

```
int valor_e = 100;
int *ptr1 = &valor_e;
int **ptr5 = &ptr1;
```

`ptr1` y `ptr5` son punteros. `ptr1` apunta a la variable `valor_e` de tipo `int`. `ptr5` contiene la dirección de `ptr1`. En la Figura 9.4 se muestran las declaraciones anteriores.

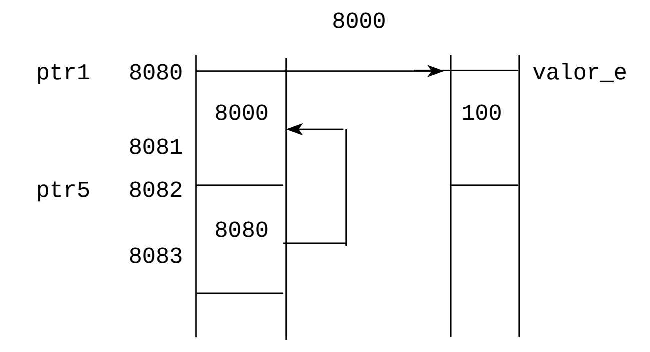


Figura 9.4. Un puntero a un puntero.

Se puede asignar valores a `valor_e` con cualquiera de las sentencias siguientes:

```
valor_e = 95;
*ptr1  = 105;         // Asigna 105 a valor_e
**ptr5  = 99;        // Asigna 99 a valor_e
```

Ejemplo

```
char c = 'z';
char* pc = &c;
char** ppc = &pc;
char*** pppc = &ppc;
***pppc = 'm';            // cambia el valor de c a 'm'
```

pppc
ppc
pc
c z

9.5. PUNTEROS Y ARRAYS

Los arrays y punteros están fuertemente relacionados en el lenguaje C++. Se pueden direccionar arrays como si fueran punteros y punteros como si fueran arrays. La posibilidad de almacenar y acceder a pun-

teros y arrays, implica que se pueden almacenar cadenas de datos en elementos de arrays. Sin punteros eso no es posible, ya que no existe el tipo de dato cadena (*string*) en C++. No existen variables de cadena, únicamente constantes de cadena.

9.5.1. Nombres de arrays como punteros

Un nombre de un array es simplemente un puntero. Supongamos que se tiene la siguiente declaración de un array:

```
int lista[5] = {10, 20, 30, 40, 50};
```

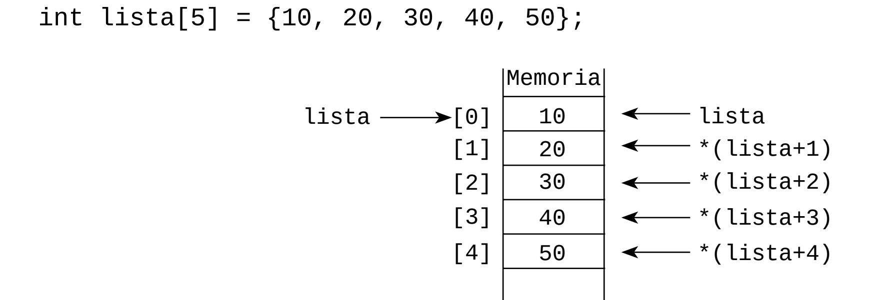


Figura 9.5. Un array almacenado en memoria.

Si se manda visualizar `lista[0]` se verá `10`. Pero, ¿qué sucederá si se manda visualizar `*lista`? Como un nombre de un array es un puntero, también se verá `10`.

Esto significa que

`lista + 0`	*apunta a*	`lista[0]`
`lista + 1`	*apunta a*	`lista[1]`
`lista + 2`	*apunta a*	`lista[2]`
`lista + 3`	*apunta a*	`lista[3]`
`lista + 4`	*apunta a*	`lista[4]`

Por consiguiente, para imprimir (visualizar), almacenar o calcular un elemento de un array, se puede utilizar notación de subíndices o notación de punteros. Dado que un nombre de un array contiene la dirección del primer elemento del array, se debe indireccionar el puntero para obtener el valor del elemento.

9.5.2. Ventajas de los punteros

Un nombre de un array es una *constante puntero*, no una variable puntero. No se puede cambiar el valor de un nombre de array, como no se pueden cambiar constantes. Esto explica porqué no se pueden asignar valores nuevos a un array durante una ejecución de un programa. Por ejemplo, si `cnombre` es un array de caracteres, la siguiente sentencia no es válida en C++:

```
cnombre = "Hermanos Daltón";
```

Se pueden asignar valores a un array sólo en tiempo de declaración, un elemento cada vez durante la ejecución o bien utilizando funciones, tales como (ya se ha hecho anteriormente) `strcpy()`.

Se pueden cambiar punteros para hacerlos apuntar a valores diferentes en memoria. El siguiente programa muestra cómo cambiar punteros. El programa define primero dos valores de coma flotante. Un puntero de coma flotante apunta a la primera variable `v1` y se utiliza en `cout`. El puntero se cambia entonces, de modo que apunta a la segunda variable de coma flotante `v2`.

```
#include <iostream.h>

int main()
{
   float v1 = 756.423;
   float v2 = 900.545;
   float *p_v;

   p_v = &v1;
   cout << "El primer valor es" << *p_v << endl;
         //se imprime 756.423

   p_v = &v2;
   cout << "El segundo valor es" << *p_v << endl;
         //se imprime 900.545;
   return 0;
}
```

Por esta facilidad para cambiar punteros, la mayoría de los programadores de C++ utilizan punteros en lugar de arrays. Como los arrays son fáciles de declarar, los programadores declaran arrays y, a continuación, utilizan punteros para referenciar dichos arrays.

9.6. ARRAYS (ARREGLOS) DE PUNTEROS

Si se necesita reservar muchos punteros a varios valores diferentes, se puede declarar un *array o* **arreglo** *de punteros*. Un array de punteros es un array que contiene elementos punteros, cada uno de los cuales apunta a un tipo de dato específico.

La línea siguiente reserva un array de diez variables puntero a enteros:

```
int *ptr[10];   // reserva un array de 10 punteros a enteros
```

La Figura 9.6 muestra cómo C++ organiza este array. Cada elemento contiene una dirección que *apunta* a otros valores de la memoria. Cada valor apuntado debe ser un entero. Se puede asignar a un elemento de `ptr` una dirección, tal como para variables puntero no arrays. Así, por ejemplo,

```
ptr[5] = &edad      // ptr[5] apunta a la dirección de edad
```

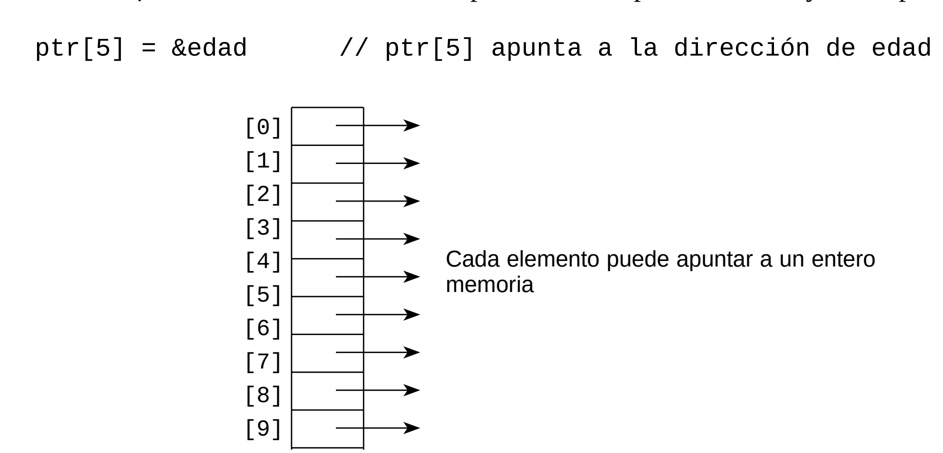


Figura 9.6. Un array de 10 punteros a enteros.

Otro ejemplo de arrays de punteros, en este caso de caracteres, es:

```
char *puntos[25];  // array de 25 punteros caracteres
```

De igual forma, se podría declarar un puntero a un array de punteros a enteros.

```
int *(*ptr10)[ ]
```

`(*ptr10)` *es un puntero a un nombre de variable*, `ptr10` *es un nombre de variable.*

```
(*ptr10)[ ]       es un puntero a un array
*(*ptr10)[ ]      es un puntero a un array de punteros a variables int
int *(*ptr10)[ ] es un puntero a un array de punteros de variables int
```

9.6.1. Inicialización de un array de punteros a cadenas

La inicialización de un array de punteros a cadenas se puede realizar con una declaración similar a ésta:

```
char *nombres_meses[12] = {  "Enero", "Febrero", "Marzo",
                              "Abril", "Mayo", "Junio",
                              "Julio", "Agosto", "Septiembre",
                              "Octubre", "Noviembre",
                              "Diciembre" };
```

9.7. PUNTEROS DE CADENAS[1]

Los punteros se pueden utilizar en lugar de índices de arrays. Considérese la siguiente declaración de un array de caracteres que contiene las veintiséis letras del alfabeto internacional (no se considera la *ñ*).

```
char alfabeto[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Declaremos ahora `p` un puntero a `char`

```
char *p;
```

Se establece que `p` apunta al primer carácter de `alfabeto` escribiendo

```
p = &alfabeto[0];         // o también p = alfabeto
```

de modo que si escribe la sentencia

```
cout << *p << endl;
```

se visualiza la letra `A`, ya que `p` apunta al primer elemento de la cadena.
Se puede hacer también

```
p = &alfabeto[15];
```

de modo que `p` apuntará al carácter decimosexto (la letra *Q*). Sin embargo, no se puede hacer

```
p = &alfabeto;
```

[1] En el Capítulo 11 se ampliará en profundidad el concepto de cadenas y su tratamiento y manipulación.

ya que `alfabeto` es un array cuyos elementos son de tipo `char`, y se produciría un error al compilar.

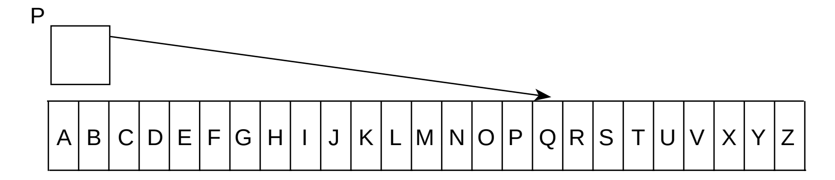


Figura 9.7. Un puntero a `alfabeto[15]`.

Es posible, entonces, considerar dos tipos de definiciones de cadena:

```
char cadena1[]="Hola viejo mundo"; //array contiene cadena
char *cptr = "C++ a su alcance";   //un puntero a la cadena
```

9.7.1. Punteros frente a arrays

El siguiente programa implementa una función para contar el número de caracteres de una cadena. En el primer programa la cadena se describe utilizando un array y en el segundo, se describe utilizando un puntero.

```
// Implementación con un array
#include <iostream>
using namespace std;

int strlen(const char cad[ ]);

void main()
{
   static char cad[ ] = "Universidad Pontificia";

   cout  << "La longitud de " << cad << " es "
         << strlen(cad) << " caracteres " << endl;
}
int strlen(const char cad[ ])
{
   int posicion = 0;
   while (cad[posicion] != '\0')
   {
      posicion++;
   }
   return posicion;
}
```

El segundo programa utiliza un puntero para contar los caracteres de la cadena.

```
#include <iostream>
using namespace std;

int strlen(const register char*);

main()
{
```

```
   static char cad[ ] = "Universidad Pontificia";

   cout  << "La longitud de " << cad << " es "
         << strlen(cad) << " caracteres " << endl;
}

int strlen(const register char* cad)
{
   int cuenta = 0;
   while (*cad++) cuenta++;
   return(cuenta);
}
```

En ambos casos se imprimirá:

```
La longitud de Universidad Pontificia es 22 caracteres
```

Comparaciones

```
int *ptr1[ ];     // Arrays de punteros a int
int (*ptr2)[ ];  // Puntero a un array de elementos int
int *(*ptr3)[ ]; // Puntero a un array de punteros a int
```

9.8. ARITMÉTICA DE PUNTEROS

Al contrario que un nombre de array, que es un puntero constante y no se puede modificar, un puntero es un puntero variable que se puede modificar. Como consecuencia, se pueden realizar ciertas operaciones aritméticas sobre punteros.

Recuérdese que un puntero es una dirección. Por consiguiente, sólo aquellas operaciones de «sentido común» son legales. Se pueden sumar o restar una constante puntero a o desde un puntero. Sumar o restar un entero. Sin embargo, no tiene sentido sumar o restar una constante de coma flotante.

Operaciones no válidas con punteros

- No se pueden sumar dos punteros.
- No se pueden multiplicar dos punteros.
- No se pueden dividir dos punteros.

Ejemplo 9.5

Si `p` apunta a la letra `'A'` en `alfabeto`, si se escribe

```
p = p+1;
```

ahora `p` apunta a la letra `'B'`.

Se puede utilizar esta técnica para explorar cada elemento de `alfabeto` sin utilizar una variable de índice. Un ejemplo puede ser

```
p = &alfa[0];
for (i = 0; i < strlen(alfa); i++)
```

```
{
   cout << *p << endl;
   p = p+1;
};
```

Las sentencias del interior del bucle se pueden sustituir por

```
cout << *p++ << endl;
```

El ejemplo anterior con el bucle `for` puede ser abreviado, haciendo uso de la característica de *terminador* nulo al final de la cadena. Utilizando la sentencia `while` para realizar el bucle y poniendo la condición de terminación de nulo o byte cero al final de la cadena. Esto elimina la necesidad del bucle `for` y su variable de control. El bucle `for` se puede sustituir por

```
while (*p) cout << *p++ << endl;
```

mientras que `*p` toma un valor de carácter distinto de cero, el bucle `while` se ejecuta, el carácter se imprime y `p` se incrementa para apuntar al siguiente carácter. Al alcanzar el byte cero al final de la cadena, `*p` toma el valor de `'\0'` o cero. El valor cero hace que el bucle termine.

9.8.1. Una aplicación de punteros

El listado `PTRMAYMI.CPP` muestra un puntero que recorre una cadena de caracteres y convierte cualquier carácter en minúsculas a caracteres mayúsculas.

```
// PTRMAYMIN.CPP
// Utiliza un puntero como índice de un array de caracteres
// y convierte caracteres minúsculas a mayúsculas

void main()
{
   char *p;
   char CadenaTexto[80];

   cout << "Introduzca cadena a convertir:";
   cin.getline(CadenaTexto, sizeof(CadenaTexto));

   // p apunta al primer carácter de la cadena
   p = &CadenaTexto[0];

   // Repetir mientras *p no sea cero
   while (*p)
      // restar 32, constante de código ASCII
      if ((*p >= 'a') && (*p <= 'z')) *p++ = *p-32;
         else p++;

   cout << "La cadena convertida es:" << endl;
   cout << CadenaTexto << endl;

   cout << "Pulse Intro(Enter) para continuar";
   cin.get();
}
```

Obsérvese que si el carácter leído cae entre 'a' y 'z', es decir, es una letra minúscula, la asignación

```
*p++ = *p-32;
```

se ejecutará, y observará que restar 32 de un código ASCII de una letra minúscula la convierte en letra mayúscula.

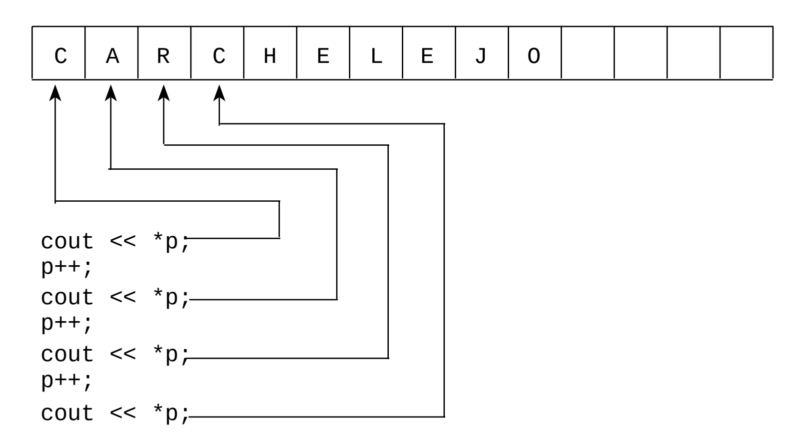


Figura 9.8. `*p++` se utiliza para acceder de modo incremental en la cadena.

9.9. PUNTEROS CONSTANTES FRENTE A PUNTEROS A CONSTANTES

Ya está familiarizado con punteros constantes, como es el caso de un nombre de un array. Un puntero constante es un puntero que no se puede cambiar, pero que los datos apuntados por el puntero pueden ser cambiados. Por otra parte, un puntero a una constante se puede modificar para apuntar a una constante diferente, pero los datos apuntados por el puntero no se pueden cambiar.

9.9.1. Punteros constantes

Para crear un puntero constante diferente de un nombre de un array, se debe utilizar el siguiente formato:

```
<tipo de dato elemento> *const <nombre puntero> =
      <dirección de variable>;
```

Como ejemplo de una definición de punteros de constantes, considérense las siguientes:

```
int x;
int y;
int *const p1 = &x;
```

`p1` es un puntero de constantes que apunta a `x`. Esto hace a `p1` una constante, pero `*p1` es una variable. Por consiguiente, se puede cambiar el valor de `*p1`, pero no `p1`. Por ejemplo, la siguiente asignación es legal, dado que se cambia el contenido de memoria a donde `p1` apunta, pero no el puntero en sí.

```
*p1 = y;
```

Por otra parte, la siguiente asignación no es legal, ya que se intenta cambiar el valor del puntero

```
p1 = &y;
```

El sistema para crear un puntero de constante a una cadena es:

```
char *const nombre = “Luis”;
```

`nombre` no se puede modificar para apuntar a una cadena diferente en memoria. Por consiguiente,

```
Strcpy (*nombre = "C");
```

es legal, ya que se modifica el dato apuntado por nombre. Sin embargo, *no es legal*

```
nombre = &Otra_Cadena;
```

dado que se intenta modificar el propio puntero.

9.9.2. Punteros a constantes

El formato para definir un puntero a una constante es

```
const <tipo de dato elemento> *<nombre puntero> =
       <dirección de const o valor real de cadena>;
```

Algunos ejemplos son:

```
const int x = 25;
const int y = 50;
const int *p1 = &x;
```

en los que `p1` se define como un puntero a la constante `x`. Los datos son constantes y no el puntero; en consecuencia, se puede hacer que `p1` apunte a otra constante.

```
p1 = &y;
```

Sin embargo, cualquier intento de cambiar el contenido almacenado en la posición de memoria a donde apunta `p1` creará un error de compilación. Así, la siguiente sentencia no se compilará correctamente:

```
*p1 = 15;
```

> ***Nota***
>
> Una definición de un puntero constante tiene la palabra reservada `const` delante del nombre del puntero, mientras que el puntero a una definición constante requiere que la palabra reservada `const` se sitúe antes del tipo de dato. Así, la definición en el primer caso se puede leer como «punteros constante o de constante», mientras que en el segundo caso la definición se lee «tipo constante de dato».

La creación de un *puntero a una constante cadena* se puede hacer del modo siguiente:

```
const char *apellido = "Mortimer";
```

9.9.3. Punteros constantes a constantes

El último caso a considerar es crear punteros constantes a constantes utilizando el formato siguiente:

```
const <tipo de dato elemento> *const <nombre puntero> =
      <dirección de const o cadena real>;
```

Esta definición se puede leer como «un tipo constante de dato y un puntero constante». Un ejemplo puede ser:

```
const int x = 25;
const int *const p1 = &x;
```

que indica: «`p1` *es un puntero constante que apunta a la constante entera* `x`». Cualquier intento de modificar `p1` o bien `*p1` producirá un error de compilación.

> ***Reglas***
>
> - Si sabe que un puntero siempre apuntará a la misma posición y nunca necesita ser reubicado (recolocado), defínalo como un puntero constante.
> - Si sabe que el dato apuntado por el puntero nunca necesitará cambiar, defina el puntero como un puntero a una constante.

Ejemplo 9.6

Un puntero a una constante es diferente de un puntero constante. El siguiente ejemplo muestra las diferencias

```
// Este trozo de código declara cuatro variables
// un puntero p; un puntero constante cp; un puntero pc a una
// constante y un puntero constante cpc a una constante

int *p;              // puntero a un int
++(*p);              // incremento int *p
++p;                 // incrementa un puntero p
int * const cp;      // puntero constante a un int
++(*cp);             // incrementa int *cp
++cp;                // no válido: el puntero cp es constante
const int * pc;      // puntero a una constante int
++(*pc);             // no válido: int * pc es constante
++pc;                // incrementa puntero pc
const int * const cpc; // puntero constante a constante int
++(*cpc);            // no válido: int *cpc es constante
++cpc;               // no válido: puntero cpc es constante
```

Regla

El espacio en blanco no es significativo en la declaración de punteros. Las declaraciones siguientes son equivalentes:

```
int* p;
int * p;
int *p;
```

9.10. PUNTEROS COMO ARGUMENTOS DE FUNCIONES

Con frecuencia, se desea que una función calcule y devuelva más de un valor, o bien se desea que una función modifique las variables que se pasan como argumentos. Cuando se pasa una variable a una función (*paso por valor*) no se puede cambiar el valor de esa variable. Sin embargo, si se pasa un puntero a una variable a una función (*paso por dirección*) se puede cambiar el valor de la variable.

Cuando una variable es local a una función, se puede hacer la variable visible a otra función pasándola como argumento. Se puede pasar un puntero a una variable local como argumento y cambiar la variable en la otra función.

Considere la siguiente definición de la función `Incrementar5`, que incrementa un entero en 5:

```
void Incrementar5(int *i)
{
   *i = *i + 5;
}
```

La llamada a esta función se realiza pasando una dirección que utilice esa función. Por ejemplo, para llamar a la función `Incrementar5` utilice:

```
int i;
i = 10;
Incrementar5(&i);
```

Es posible mezclar paso por referencia y por valor. Por ejemplo, la función `func1` definida como

```
int func1(int *s, int t)
{
   *s = 6;
    t = 25;
}
```

y la invocación a la función podría ser:

```
int i, j;
i = 5;
j = 7;
func1(&i, j);      //llamada a func1
```

Cuando se retorna de la función `func1` tras su ejecución, `i` será igual a `6` y `j` seguirá siendo `7`, ya que se pasó por valor.

El paso de un nombre de array a una función es lo mismo que pasar un puntero al array. Se pueden cambiar cualquiera de los elementos del array. Cuando se pasa un elemento a una función, sin embargo, el elemento se pasa por valor. En el ejemplo

```
int lista[] = {1, 2, 3};
func(lista[1], lista[2]);
```

ambos elementos se pasan por valor.

> *Los parámetros dirección son más comunes en C, dado que en C++ existen los parámetros por referencia que resuelven mejor la modificación de los parámetros dentro de funciones.*

9.10.1. Paso por referencia frente a paso por dirección

Aunque el paso por referencia es más eficiente que el paso por dirección, vamos a mostrar las diferencias con un ejemplo. Supongamos que se crea una estructura para registrar las temperaturas más alta y más baja de un día determinado.

```
struct temperatura {
float alta;
float baja;
};
```

Un caso típico podría ser almacenar las lecturas de un termómetro conectado de algún modo posible a una computadora. Una función clave del programa lee la temperatura actual y modifica el miembro adecuado, `alta` o `baja`, en una estructura `temperatura` pasada por referencia a la función.

Método C++

```
void registrotemp(temperatura &t)
{
   float actual;

   leertempactual(actual);
   if (actual > t.alta)
      t.alta = actual;
   else if (actual < t.baja)
      t.baja = actual;
}
```

Como el parámetro de `registrotemp` se pasa por referencia, la función actúa directamente sobre la variable pasada, por ejemplo `temp`, mediante las sentencias

```
temperatura temp;
registrotemp(temp);
```

Método C

En C todos los parámetros se pasan por valor; en consecuencia, para referirse a un argumento real que se desea modificar habrá que pasar la dirección del argumento a un puntero parámetro. La misma función `registrotemp` escrita en C sería:

```
void registrotemp(struct temperatura *t)
{
```

```
   float actual;

   leertempactual(actual);
   if (actual > t -> alta)
      t -> alta = actual;

   else if (actual < t -> baja)
      t -> baja = actual;
}
```

9.11. PUNTEROS A FUNCIONES

Hasta este momento se han analizado punteros a datos. Es posible declarar punteros a cualquier tipo de variables, estructura o array. De igual modo, las funciones pueden declarar parámetros punteros para permitir qué sentencias pasen las direcciones de los argumentos a esas funciones.

Es posible también crear punteros que apunten a funciones. En lugar de direccionar datos, los punteros de funciones apuntan a código ejecutable. Al igual que los datos, las funciones se almacenan en memoria y tienen direcciones iniciales. En C++ (y en C) se pueden asignar las direcciones iniciales de funciones a punteros. Tales funciones se pueden llamar en un modo indirecto, es decir, mediante un puntero cuyo valor es igual a la dirección inicial de la función en cuestión.

La sintaxis general para la declaración de un puntero a una función es:

```
Tipo_de_retorno (*PunteroFunción) (<lista de parámetros>);
```

Este formato indica al compilador que `PunteroFunción` es un puntero a una función que devuelve el tipo `Tipo_de_retorno` y una lista de parámetros.

Un puntero a una función es simplemente un puntero cuyo valor es la dirección del nombre de la función. Dado que el nombre es, en sí mismo, un puntero, un puntero a una función es un puntero a un puntero constante.

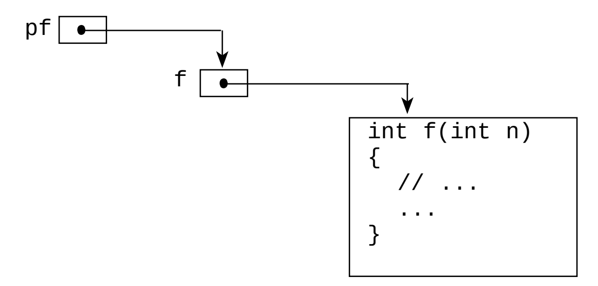


Figura 9.9. Puntero a función.

Por ejemplo:

```
int f(int);              // declara la función f
int (*pf)(int);          // declara el puntero pf a la función
pf = &f;                 // asigna la dirección de f a pf
```

Ejemplo 9.7

```
double (*fp) (int n);
float (*p) (int i, int j);
```

```
void (*sort) (int* ArrayEnt, unsigned n);
unsigned (*search)(int BuscarClave,int* ArrayEnt,unsigned n);
```

El primer identificador, `fp`, apunta a una función que devuelve un tipo `double` y tiene un único parámetro de tipo `int`. El segundo puntero, `p`, apunta a una función que devuelve un tipo `float` y acepta dos parámetros de tipo `int`. El segundo puntero, `sort`, es un puntero a una función que devuelve un tipo `void` y toma dos parámetros: un puntero a `int` y un tipo `unsigned`. Por último, `search` es un puntero a una función que devuelve un tipo `unsigned` y tiene tres parámetros: un `int`, un puntero a un `int` y un `unsigned`.

9.11.1. Inicialización de un puntero a una función

La sintaxis general para inicializar un puntero a una función es:

PunteroFunción = unaFunción

La función asignada debe tener el mismo tipo de retorno y lista de parámetros que el puntero a función; en caso contrario, se producirá un error de compilación. Así, por ejemplo, un puntero `ordenar` a una función:

```
void (*ordenar) (int* Lista, unsigned n);
ordenar = qsort;    // inicialización de ordenar
```

Algunas de las funciones de la biblioteca en tiempo de ejecución, tales como `qsort()`, requiere pasar un argumento que consta de un puntero a una función. Se debe pasar a `qsort` un puntero de función que apunta a una función.

Ejemplo 9.8

Supongamos un puntero p a una función tal como

```
float (*p) (int i, int j);
```

a continuación se puede asignar la dirección de la función `ejemplo`:

```
float ejemplo(int i, int j)

{
   return 3.14159 * i * i + j;
}
```

al puntero `p` escribiendo

```
p = ejemplo;
```

Después de esta asignación se puede escribir la siguiente llamada a la función:

```
(*p) (12, 45)
```

Su efecto es el mismo que

```
ejemplo (12, 45)
```

También se puede omitir el asterisco (así como los paréntesis) en la llamada `(*p) (12, 45)`, que proporciona

```
p (12, 45)
```

La *utilidad de las funciones a punteros* se ve más claramente si se imagina un programa grande, al principio del cual se desea elegir una entre varias funciones, de modo que la función elegida se llama, entonces, muchas veces. Mediante un puntero, la elección sólo se hace una vez: después de asignar (la dirección de) la función seleccionada a un puntero y después se puede llamar a través de ese puntero.

Los punteros a funciones también permiten pasar una función como un argumento a otra función. Para pasar el nombre de una función como un argumento función, se especifica el nombre de la función como argumento. Supongamos que se desea pasar la función mifunc() a la función sufunc(). El código siguiente realiza las tareas anteriores:

```
void sufunc(int (*f) ());             // prototipo de sufunc
int mifunc(int i);                    // prototipo de mifunc
main()
{
   ...
   sufunc(mifunc);
}

int mifunc(int i)
{
   return 5*i;
}
```

En la función llamada se declara la función pasada como un puntero función.

```
void sufunc(int (*f) ())
{
   ...
      j = f(5);
   ...
}
```

Como ejemplo práctico veamos cómo escribir una función general que calcule la suma de algunos valores, es decir,

```
f(1) + f(2) + ... + f(n)
```

para cualquier función `f` que devuelva el tipo `double` y con un argumento `int`. Diseñaremos una función `funcsuma` que tiene dos argumentos: *n*, el número de términos de la suma, y *f*, la función utilizada. Así pues, la función `funcsuma` se va a llamar dos veces, y va a calcular la suma de

```
inversos(k) = 1.0 / k    (k = 1, 2, 3, 4, 5)
cuadrados(k) = k²    (k = 1, 2, 3)
```

El programa `PUNTFUN1.CPP` muestra la función `funcsuma`, que utiliza la función `f` en un caso inversos y en otro cuadrados.

```
// PUNTFUN1.CPP
#include <iostream.h>
```

```
using name space std;

double inversos(int k);
double cuadrados(int k);
double funcsuma(int n, double (*f) (int k));

int main()
{
   cout  << "Suma de cinco inversos:"
         << funcsuma(5, inversos) << endl;
   cout  << "Suma de tres cuadrados:"
         << funcsuma(3, cuadrados) << endl;
   return 0;
}

double funcsuma(int n, double (*f) (int k))
{
   double s = 0;
   int i;
   for (i = 1; i <= n; i++)
      s += f(i);
   return s;
}

double inversos(int k)
{
   return 1.0 / k;
}

double cuadrados(int k)
{
   return (double)k * k;
}
```

El programa anterior calcula las sumas de

a) $1 + \frac{1.0}{2} + \frac{1.0}{3} + \frac{1.0}{4} + \frac{1.0}{5}$

b) 1.0 + 4.0 + 9.0

y su salida será:

```
Suma de cinco inversos: 2.283333
Suma de tres cuadrados: 14
```

9.11.2. Otra aplicación

Algunas de las funciones de la biblioteca en tiempo de ejecución, tal como `qsort()`, requieren pasar un argumento que consta de un puntero a una función. Se debe pasar a qsort un puntero de función que apunta hacia una función que se indica y se ha de proporcionar. `qsort()` utiliza el algoritmo de ordenación rápida (*quicksort*) para ordenar un array de cualquier tipo de dato. Se debe proporcionar una

función para comparar las relaciones de elementos de arrays. En el programa `PUNTFUN2.CPP` la función `comparar()` se pasa a `qsort()`. La función `comparar()` compara entradas del array `tabla` y devuelve (retorna) un número negativo si `arg1` es menor que `arg2`, devuelve cero si son iguales, o un número positivo si `arg1` es mayor que `arg2`.

```
// PUNTFUN2.CPP. Ordena utilizando el algoritmo quicksort
#include <iostream>
#include <search.h>       // archivo de cabecera
using namespace std;

int comparar(const void *arg1, const void *arg2);
void main()
{
   int i, j, aux, text;
   int tabla[10] = {14, 17, 5, 21, 12, 40, 60, 55, 35, 15};

   // Ordena tabla utilizando el algoritmo quicksort
   // de la biblioteca en tiempo de ejecución
   qsort((void *) tabla, (size_t)10, sizeof(int), comparar);

   cout << "Lista ordenada: \n";
   for (i = 0; i < 10; i++)
      cout << " " << tabla[i];
   return;
}
// Comparar dos elementos de la lista
int comparar(const void *arg1, const void *arg2)
{
   return *(int *) arg1 - *(int *) arg2;
}
```

La salida del programa es:

```
Lista ordenada: 5  12  14  15  17  21  35  40  55  65
```

Recuerde

Los parámetros de la función `qsort()` son:

- `(void *) lista` Array que contiene valores a ordenar.
- `(size_t) 10` Número de elementos del array.
- `sizeof(int)` Tamaño en bytes de cada elemento del array.
- `comparar` Nombre de la función que compara dos elementos del array.

9.11.3. Arrays de punteros de funciones

Ciertas aplicaciones requieren disponer de numerosas funciones, basadas en el cumplimiento de ciertas condiciones. Un método para implementar tal aplicación es utilizar una sentencia switch con muchas sentencias `case`. Otra solución es utilizar un array de punteros de función. Se puede seleccionar una función de la lista y llamarla.

La sintaxis general de un array de punteros de función es:

```
tipoRetorno(*PunteroFunc[LongArray])(<Lista de parámetros>);
```

Ejemplo 9.9

```
double (*fp[3]) (int n);
void (*ordenar[MAX_ORD]) (int* ArrayEnt, unsigned n);
```

`fp` apunta a un array de funciones; cada miembro devuelve un valor `double` y tiene un único parámetro de tipo `int.` `ordenar` es un puntero a un array de funciones; cada miembro devuelve un tipo `void` y toma dos parámetros: un puntero a `int` y un `unsigned`.

Recuerde

- `func`, nombre de un objeto.
- `func[]` es un array.
- `(*func[])` es un array de punteros.
- `(*func[])()` es un array de punteros a funciones.
- `int (*func[])()` es un array de punteros a funciones que devuelven variables `int`.

Se puede asignar la dirección de las funciones al array, proporcionando las funciones que ya han sido declaradas. Un ejemplo es

```
int func1(int i, int j);
int func2(int i, int j);
int (*func[])() = {func1, func2};
```

9.11.4. Una aplicación práctica

El listado `CALCULA1.CPP` es un programa calculador que puede sumar, restar, multiplicar o dividir números. Se escribe una expresión simple por teclado y el programa visualiza la respuesta.

```
// CALCULA1.CPP. Ilustra uso de punteros a funciones
#include <iostream>
using namespace std;

float sumar(float x, float y);
float restar(float x, float y);
float mult(float x, float y);
float div(float x, float y);
float (*f) (float x, float y);

void main()
{
   char signo, operadores[] = {'+', '-', '*', '/'};
   float(*func[])(float, float) = {sumar, restar, mult, div};
   int i;
   float x, y, z;
```

```
   cout << "     Calculador\n Expresión:";
   cin >> x >> signo >> y;
   for (i = 0; i < 4; i++)
   {
      if (signo == operadores[i])
      {
         f = func[i];
         z = f(x, y);
         cout  << "\n" << x
               << " "  << signo
               << " "  << y
               << " "  << z;
      }
   }
}

float sumar(float x, float y)
{
   return x + y;
}

float restar(float x, float y)
{
   return x - y;
}

float mult(float x, float y)
{
   return x * y;
}

float div(float x, float y)
{
   return x / y;
}
```

9.12. PUNTEROS A ESTRUCTURAS

Un puntero también puede apuntar a una estructura. Se puede declarar un puntero a una estructura tal como se declara un puntero a cualquier otro objeto.

Se declara un puntero estructura tal como se declara cualquier otra variable estructura: especificando un puntero en lugar del nombre de la variable estructura.

```
struct persona
{
   char nombre[30];
   int edad;
   int altura;
   int peso;
};
persona empleado = {"Mortimer, Pepe", 47, 182, 85"};
```

```
persona *p;          // se crea un puntero de estructura
p = &persona;
```

Cuando se referencia un miembro de la estructura utilizando el nombre de la estructura, se especifica la estructura y el nombre del miembro separado por un punto (.). Para referenciar el nombre de una persona, utilice `empleado.nombre.`

Cuando se referencia una estructura utilizando el puntero estructura, se emplea el operador ->.

Ejemplo 9.10

```
// PUNTEST1.CPP
#include <iostream>
using namespace std;

struct persona
{
   char nombre[30];
   int edad;
   int altura;
   int peso;
};

void mostrar_persona(persona *ptr);
void main()
{
   int i;
   persona empleados[] = { {"Mortimer, Pepe", 47, 182, 85"},
                         {"García, Luis", 39, 170, 75"},
                         {"Jiménez, Tomás",18, 175, 80"} };
   persona *p;    // puntero a estructura
   p = empleados;
   for (i = 0; i < 3; i++, p++);
      mostrar_persona(p);
}

void mostrar_persona(persona *ptr)
{
   cout  << "Nombre:     " << ptr -> nombre
         << "Edad:       " << ptr -> edad
         << "Altura:     " << ptr -> altura
         << "Peso:       " << ptr -> peso << "\n";
}
```

Al ejecutar este programa se visualiza la salida siguiente:

```
Nombre: Mortimer, Pepe   Edad: 47  Altura: 180  Peso: 85
Nombre: García, Luis     Edad: 39  Altura: 170  Peso: 75
Nombre: Jiménez, Tomás   Edad: 18  Altura: 175  Peso: 80
```

RESUMEN

Los punteros son una de las herramientas más eficientes para realizar aplicaciones en C++. Aunque su práctica puede resultar difícil y tediosa es, sin lugar a dudas, una necesidad vital su aprendizaje si desea obtener el máximo rendimiento de sus programas.

En este capítulo habrá aprendido los siguientes conceptos:

- Un puntero es una variable que contiene la dirección de una posición en memoria.
- Para declarar un puntero se sitúa un asterisco entre el tipo de dato y el nombre de la variable, como en `int *p`.
- Para obtener el valor almacenado en la dirección utilizada por el puntero, se utiliza el operador de indirección (`*`). El valor de `p` es una dirección de memoria y el valor de `*p` es la cantidad almacenada en esa dirección de memoria.
- Para obtener la dirección de una variable existente, se utiliza el operador de dirección (`&`).
- Se debe declarar un puntero antes de su uso.
- Un puntero `void` es un puntero que no se asigna a un tipo de dato específico y puede, por consiguiente, utilizarse para apuntar a tipos de datos diferentes en diversos lugares de su programa.
- Para inicializar un puntero que no apunta a nada, se utiliza la constante `NULL`.
- Estableciendo un puntero a la dirección del primer elemento de un array se puede utilizar el puntero para acceder a cada elemento del array de modo secuencial.

Asimismo, se han estudiado los conceptos de aritmética de punteros, punteros a funciones, punteros a estructuras y arrays de punteros.

EJERCICIOS

9.1. Si `p` y `q` son punteros a estructuras de tipo `electrico`, explicar el efecto de cada sentencia válida. ¿Cúales no son válidas?

```
struct electrico {
  string corriente;
  int voltios;
};

electrico  *p, *q;
```

a. `p -> corriente = "CA";`
b. `p -> voltios = q -> voltios;`
c. `*p = *q;`
d. `p = q ;`
e. `p -> corriente _ "AT";`
f. `p -> corriente = q -> voltios ;`
g. `p = 53;`
h. `*q = p;`

9.2. Si `a`, `b` y `c` son punteros a estructuras de tipo `electrico`, dibujar un diagrama de punteros y celdas de memoria después de las operaciones siguientes:

a. `a = new  electrico ;`
d. `a = b ;`
b. `b = new electrico ;`
e. `b = c ;`
c. `c = new electrico ;`
f. `c = a ;`

9.3. ¿Son iguales o diferentes las direcciones de memoria de `r` y `n`? ¿Por qué?

```
void main()
{
  int n = 33;
  int &r = n;
  cout << "&n = “ << &n << ", &r =
  " << &r << endl;
}
```

9.4. ¿Cuál es la salida del siguiente programa?

```
void main()
{
  int n = 35;
  int *p = &n;
  int &r = *p;
  cout << "r = " << r << endl;
}
```

9.5. ¿Cuál es la salida del siguiente programa?

```
void main ()
{
  int n = 35;
  int *p = fn;
  cout << "*p = " << *p << endl;
}
```

9.6. ¿Cuál es la salida del siguiente programa que utiliza una función que devuelve un array?

```
float &componente(float *v, int k)
{
  return v[k-1];
}

void main()
{
  float v[5];
  for (int k = 1; k <= 5; k++)
    componente(v,k) = 1.0/k;
  for (int i = 0; i < 5; i++)
     cout << "v[" << i<< "] = " <<
   v[i] << endl;
}
```

9.7. Indicar cuál de las siguientes sentencias son válidas y cuáles no razonando la respuesta e indicando cuál es la tarea que realizan.

```
int *p;
++(*p);
++p;
int * const cp;
++(*cp);
++cp;
const int *pc;
++(*pc);
const int * const cpc;
++(*cpc);
++cpc;
```

9.8. Explique y dibuje un esquema gráfico que muestre las variables y direcciones apuntadas por las siguientes sentencias:

```
char c = 'v';
char *pc-; = &c;
char **ppc = &pc;
char ***pppc = &ppc;
***pppc = 'w';
```

9.9. Explique y dibuje con un esquema la misión de las siguientes declaraciones:

```
int g(int);
int (*pg)(int);
pf = &g;
```

PROBLEMAS

9.1. Escribir un segmento de programa que cree una colección de siete punteros al tipo estructura `nota_musical`, asigne memoria para cada puntero y sitúe las notas musicales *do, re, mi, fa, so, la, si* en las áreas de datos.

```
struct  nota_musical
{
  nota-:  string-;
};
```

9.2. Escribir un programa que muestre cómo un puntero se puede utilizar para recorrer un array.

9.3. Escribir una función que utilice punteros para copiar un array de datos de tipo `double`.

9.4. Escribir una función que utilice punteros para buscar la dirección de un entero dado en un array dado. Si se encuentra el entero dado, la función devuelve su dirección; en caso contrario, devuelve `NULL`.

9.5. Escribir una función a la cual se pase un array de `n` punteros a `float` y devuelva un array que se ha creado de nuevo y que contenga esos `n` valores `float`.

9.6. Escribir una función `suma` mediante punteros a funciones que calcule la suma de los cuadrados y de los cubos de los *n* primeros números enteros.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 219).
2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

9.1. Encontrar los errores de las siguientes declaraciones de punteros:

```
int x, *p, &y;
char* b= "Cadena larga";
char* c= 'C';
float x;
void* r = &x;
```

9.2. ¿Qué diferencias se pueden encontrar entre un puntero a constante y una constante puntero?

9.3. Un array unidimensional se puede indexar con la aritmética de punteros. ¿Qué tipo de puntero habría que definir para indexar un array bidimensional?

9.4. El código siguiente accede a los elementos de una matriz. Acceder a los mismos elementos con aritmética de punteros.

```
#define N 4
#define M 5
int f,c;
double mt[N][M];
     . . .
for (f = 0; f < N; f++)
{
   for (c = 0; c < M; c++)
         cout << mt[f][c];
   cout << "\n";
}
```

9.5. Escribir una función con un argumento de tipo puntero a `double` y otro argumento de tipo `int`. El primer argumento se debe de corresponder con un array y el segundo con el número de elementos del array. La función ha de ser de tipo puntero a `double` para devolver la dirección del elemento menor.

9.6. Dada las siguientes definiciones y la función `gorta`:

```
double W[15], x, z;
void *r;
double* gorta(double* v, int m, dou-
   ble k)
{
 int j;
 for (j = 0; j < m; j++)
 if (*v == k)
    return v;
 return 0,
}
```

- ¿Hay errores en la codificación? ¿De qué tipo?
- ¿Es correcta la siguiente llamada a la función?:

```
r = gorta(W,10,12.3);
```

- ¿Y estas otras llamadas?:

```
cout << (*gorta(W,15,10.5));
z = gorta(w,15,12.3);
```

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág, 222).
2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

9.1. Se quiere almacenar la siguiente información de una persona: nombre, edad, altura y peso. Escribir una función que lea los datos de una persona, recibiendo como parámetro un puntero y otra función que los visualice.

9.2. Escribir un programa que decida si una matriz de números reales es simétrica. Utilizar: 1, una función de tipo `bool` que reciba como entrada una matriz de reales, así como el número de filas y de columnas, y decida si la matriz es simétrica; 2, otra función que genere la matriz de 10 filas y 10 columnas de números aleatorios de 1 a 100; 3, un programa principal que realice llamadas a las dos funciones. ***Nota***: usar la transmisión de matrices como parámetros punteros y la aritmética de punteros para la codificación.

9.3. Escribir un programa para generar una matriz de 4 × 5 números reales, y multiplique la primera columna por otra cualquiera de la matriz y mostrar la suma de los productos. El programa debe descomponerse en subprogramas y utilizar punteros para acceder a los elementos de la matriz.

9.4. Codificar funciones para sumar a una fila de una matriz otra fila; intercambiar dos filas de una matriz, y sumar a una fila una combinación lineal de otras. ***Nota*:** debe realizarse con punteros y aritmética de punteros.

9.5. Codificar funciones que realicen las siguientes operaciones: multiplicar una matriz por un número, y rellenar de ceros una matriz. ***Nota*:** usar la aritmética de punteros.

9.6. Escribir un programa en el que se lea 20 líneas de texto, cada línea con un máximo de 80 caracteres. Mostrar por pantalla el número de vocales que tiene cada línea, el texto leído, y el número total de vocales leídas.

9.7. En una competición de natación se presentan 16 nadadores. Cada nadador se caracteriza por su nombre, edad, prueba en la que participa y tiempo (minutos, segundos) de la prueba. Escribir un programa que realice la entrada de los datos y calcule la desviación típica respecto al tiempo.

9.8. Escribir un programa que permita calcular el área de diversas figuras: un triángulo rectángulo, un triángulo isósceles, un cuadrado, un trapecio y un círculo. ***Nota*:** utilizar un array de punteros de funciones, siendo las funciones las que permiten calcular el área.

9.9. Desarrolle un programa en C++ que use una estructura para la siguiente información sobre un paciente de un hospital: nombre, dirección, fecha de nacimiento, sexo, día de visita y problema médico. El programa debe tener una función para entrada de los datos de un paciente, guardar los diversos pacientes en un array y mostrar los pacientes cuyo día de visita sea uno determinado.

9.10. Escribir una función que tenga como entrada una cadena y devuelva un número real. La cadena contiene los caracteres de un número real en formato decimal (por ejemplo, la cadena "25.56" se ha de convertir en el correspondiente valor real).

9.11. Escribir un programa para simular una pequeña calculadora que permita leer expresiones enteras terminadas en el símbolo = y las evalúe de izquierda a derecha sin tener en cuenta la prioridad de los operadores. Ejemplo 4*5-8=12. ***Nota*:** usar un array de funciones para realizar las distintas operaciones (suma, resta, producto, cociente resto).

CAPÍTULO 10

Asignación dinámica de memoria

Contenido

INTRODUCCIÓN

Los programas pueden crear variables globales o locales. Las variables declaradas globales en sus programas se almacenan en posiciones fijas de memoria, en la zona conocida como *segmento de datos* del programa, y todas las funciones pueden utilizar estas variables. Las variables locales se almacenan en la **pila** (*stack*) y existen *sólo* mientras son activas las funciones que están declaradas. Es posible, también, crear variables `static` (estáticas a las globales) que se almacenan en posiciones fijas de memoria, pero sólo están disponibles en el módulo (es decir, el archivo de texto) o función en que se declaran; su espacio de almacenamiento es el segmento de datos.

Todas estas clases de variables comparten una característica común: se definen cuando se compila el programa. Esto significa que el compilador reserva (define) espacio para almacenar valores de los tipos de datos declarados. Es decir, en el caso de las variables globales y locales se ha de indicar al compilador exactamente cuántas y de qué tipo son las variables a asignar. O sea, el espacio de almacenamiento se reserva en el momento de la compilación.

Sin embargo, no siempre es posible conocer con antelación a la ejecución cuánta memoria se debe reservar al programa.

En C++ se asigna memoria en el momento de la ejecución en el *montículo o montón* (**heap**), mediante las funciones `malloc()` y `free()`. C++ ofrece un método mejor y más eficiente que C para realizar la gestión dinámica (en tiempo de ejecución) de la memoria. En lugar de llamar a una función para asignar o liberar memoria, C++ invoca un operador. C++ proporciona dos operadores para la gestión de la memoria: `new()` y `delete()`, que asignan y liberan la memoria de una zona de memoria denominada *almacén libre* (**free store**).

CONCEPTOS CLAVE

- Array dinámico.
- Array estático.
- Desbordamiento de memoria.
- Estilo C (`free`).
- Estilo C (`malloc`).
- Gestión dinámica.
- Operador `delete`.
- Operador `new`.

10.1. GESTIÓN DINÁMICA DE LA MEMORIA

La estructura de la memoria principal de la computadora se divide esencialmente en: memoria estática ocupada para las variables y la memoria dinámica que se gestiona con los operadores `new` y `de1ete`, y que se denomina también *montículo* o *montón (heap)*. La memoria dinámica funciona mediante la reserva y liberación de trozos de memoria a voluntad del programador. En ausencia de un *"recolector automático de basura" —garbage recollector—* (como existe en otros lenguajes de programación, tales como Java) la memoria dinámica permanece ocupada hasta que el programador la libera explícitamente, devolviéndola a la zona de memoria dinámica libre, donde puede volver a ser utilizada. En C no existía ningún operador para la reserva y liberación de memoria dinámica y su manejo se efectuaba con las funciones `malloc`, `calloc` y `free`. En C++, existe una alternativa mejor, los operadores `new` y `delete`. La reserva de memoria (el tamaño) se realiza con `new` y la liberación cuando ya no se necesita se realiza con el operador `de1ete`, que deja libre el espacio ocupado para una posterior utilización. De igual forma `new` asigna memoria para un objeto y `delete` destruye y libera la memoria ocupada por el objeto.

Consideremos un programa que evalúe las calificaciones de los estudiantes de una asignatura. El programa almacena cada una de las calificaciones en los elementos de una lista o tabla (*array*). El tamaño del array debe ser lo suficientemente grande para contener el total de alumnos matriculados en la asignatura. La sentencia

```
int asignatura [40];
```

reserva, por ejemplo, 40 enteros. Los arrays son un método muy eficaz cuando se conoce su longitud o tamaño en el momento de escribir el programa. Sin embargo, presentan un grave inconveniente si el tamaño del array *sólo* se conoce en el momento de la ejecución. Las sentencias siguientes producirían un error durante la compilación:

```
cin >> num_estudiantes;
int asignatura[num_estudiantes];
```

ya que el compilador requiere que el tamaño del array sea constante. Sin embargo, en numerosas ocasiones no se conoce la memoria necesaria hasta el momento de la ejecución. Por ejemplo, si se desea almacenar una cadena de caracteres tecleada por el usuario, no se puede prever, *a priori*, el tamaño del array necesario, a menos que se reserve un array de gran dimensión y se malgaste memoria cuando no se utilice. En el ejemplo anterior, si el número de alumnos de la clase aumenta, se debe variar la longitud del array y volver a compilar el programa. El método para resolver este inconveniente es recurrir a *punteros* y a técnicas de *asignación dinámica de memoria*.

Un espacio de la variable asignada dinámicamente se crea durante la ejecución del programa, al contrario que en el caso de una variable local cuyo espacio se asigna en tiempo de compilación. La asignación dinámica de memoria proporciona control directo sobre los requisitos de memoria de su programa. El programa puede crear o destruir la asignación dinámica en cualquier momento durante la ejecución. Se puede determinar la cantidad de memoria necesaria en el momento en que se haga la asignación. Dependiendo del modelo de memoria en uso, se pueden crear variables mayores de 64 K.

El código del programa compilado se sitúa en segmentos de memoria denominados *segmentos de código*. Los datos del programa, tales como variables globales, se sitúan en un área denominada *segmento de datos*. Las variables locales y la información de control del programa se sitúan en un área denominada *pila*. La memoria que queda se denomina *memoria del montículo* o *almacén libre*. Cuando el programa solicita memoria para una variable dinámica, se asigna el espacio de memoria deseado desde el montículo.

ERROR TÍPICO DE PROGRAMACIÓN EN C/C++

La declaración de un array exige especificar su longitud como una expresión constante, así s declara un array de 100 elementos:

```
char s[100];
```

Si se utiliza una variable o una expresión que contiene variables se producirá un error.

```
int n;
  ...
  cin >> n;
  char s[n];            // error
```

10.1.1. Almacén libre (*free store*)

El mapa de memoria del modelo de un programa grande es muy similar al mostrado en la Figura 10.1. El diseño exacto dependerá del modelo de programa que se utilice. Para grandes modelos de datos, el almacén libre (*heap*) se refiere al área de memoria que existe dentro de la pila del programa. Y el almacén libre es, esencialmente, toda la memoria que queda libre después de que se carga el programa.

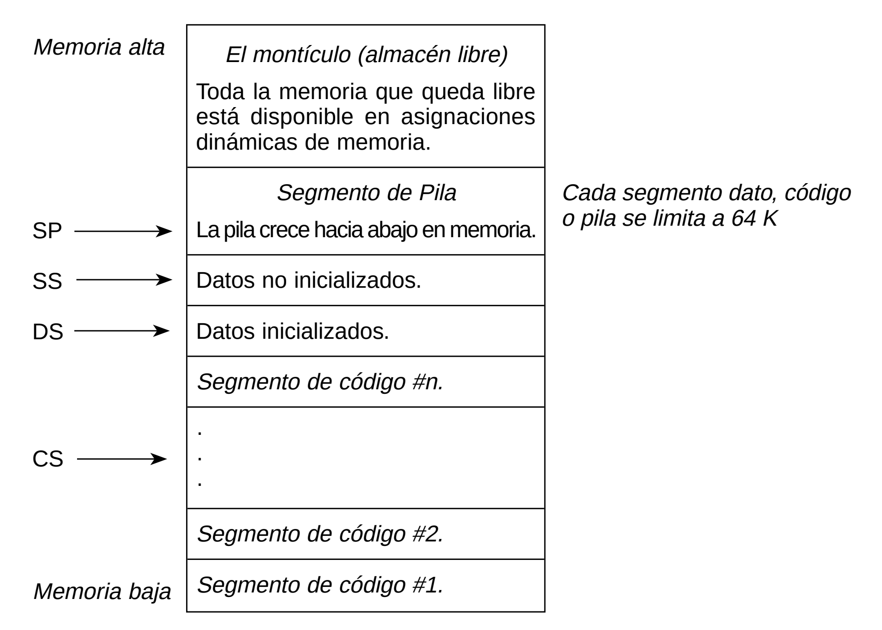


Figura 10.1. Mapa de memoria de un programa.

10.1.2. Ventajas de la asignación dinámica de memoria en C++

En C las funciones **`malloc()`** y **`free()`** asignan y liberan memoria de un bloque de memoria denominado el *montículo del sistema*. La función `malloc()` asigna memoria utilizando *asignación dinámica* debido a que puede gestionar la memoria utilizando la ejecución de un programa.

C++ ofrece un nuevo y mejor método para gestionar la asignación dinámica de memoria, los operadores `new` y `delete`, que asignan y liberan la memoria de una zona de memoria llamada *almacén libre*.

Los operadores `new` y `delete` son más versátiles que **`malloc()`** y **`free()`**, ya que ellos pueden asociar la asignación de memoria con el medio que lo utiliza. Son más fiables, ya que el compilador realiza verificación de tipos cada vez que un programa asigna memoria con `new`.

Otra ventaja proviene del hecho de que `new` y `delete` se implementan como operadores y no como funciones. Esto significa que `new` y `delete` se construyen en el propio lenguaje de modo que los programas pueden utilizar `new` y `delete` sin incluir ningún archivo de cabecera.

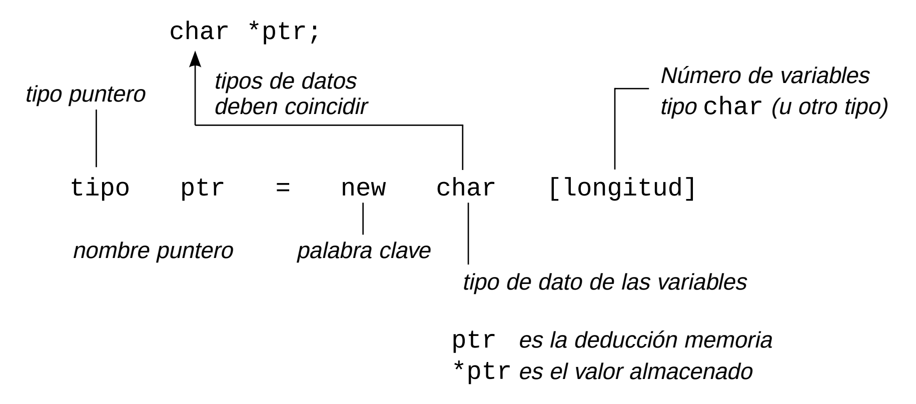
Existe otra característica importante de los operadores `new` y `delete` y es que ellos nunca requieren *moldeado* (conversión forzosa de tipos) de tipos y eso hace a `new` y `delete` más fáciles de utilizar que **`malloc()`** y **`free()`**.

Los objetos creados con `new` residen en el *almacenamiento libre*. Los objetos del almacenamiento libre se eliminan con el operador `delete`.

10.2. EL OPERADOR *new*

C++ proporciona otro método para obtener bloques de memoria: el operador **new**. El operador `new` asigna un bloque de memoria que es el tamaño del tipo de dato. El dato u objeto dato puede ser un `int`, un `float`, una estructura, un array o cualquier otro tipo de dato. El operador `new` devuelve un puntero, que es la dirección del bloque asignado de memoria. El puntero se utiliza para referenciar el bloque de memoria.

Sintaxis completa del operador new



Ejemplo

```
char * p = new char [100];
```

El formato práctico del operador `new` es:

puntero = `new` *nombreTipo (inicializador opcional)*

o bien

```
1. tipo *puntero = new tipo                  // No arrays
2. tipo *puntero = new tipo[dimensiones]     // Arrays
```

Cada vez que se ejecuta una expresión que invoca al operador `new`, el compilador realiza una verificación de tipo para asegurar que el tipo del puntero especificado en el lado izquierdo del operador es el tipo correcto de la memoria que se asigna en la derecha. Si los tipos no coinciden, el compilador produce un mensaje de error.

El formato 1 es para tipos de datos básicos y estructuras, y el segundo es para arrays. El elemento *tipo* se especifica dos veces y ambos deben ser del mismo tipo. La primera instancia de *tipo* define el tipo de datos del puntero y la segunda instancia define el tipo de dato del objeto. El campo *puntero* es el nombre del puntero al que se asigna la dirección del objeto dato, o `NULL`, si falla la operación de asignación de memoria. En el segundo formato, al *puntero* se le asigna la dirección de memoria de un bloque lo suficientemente grande para contener un array con *dimensiones* elementos.

Sintaxis de los operadores `new` y `delete`

puntero = `new` *tipo ;*

`delete` puntero ;

El operador `new` devuelve la dirección de la variable asignada dinámicamente. El operador `delete` elimina la memoria asignada dinámicamente a la que se accede por un puntero.

Sintaxis de asignación-desasignación de un array dinámico

Puntero Array = `new` *[tamaño Array];*
`delete[]` *Puntero Array ;*

Figura 10.2. Sintaxis (formato) del operador new.

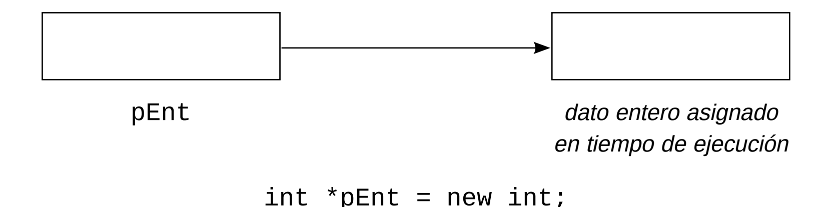
El siguiente código utiliza `new` para asignar espacio para un valor entero:

```
int *pEnt;
...
pEnt = new int;
```

Se pueden combinar las dos sentencias en una sola

```
int *pEnt = new int;
```

El efecto de esta sentencia es crear una variable entera sin nombre, accesible sólo a través de un puntero `pEnt`



`int *pEnt = new int;`

La llamada a `new` asigna espacio para un `int` (entero) y almacena la dirección de la asignación en `pEnt. pEnt` apunta ahora a la posición en el almacén libre (montículo) donde se establece la memoria. La Figura 10.3 muestra cómo `pEnt` apunta a la asignación del almacén libre.

Así por ejemplo, para reservar memoria para un array de 100 enteros:

```
int *BloqueMem;
BloqueMem = new int[100];
```

o bien, de un modo más conciso

```
int *BloqueMem = new int[100];
```

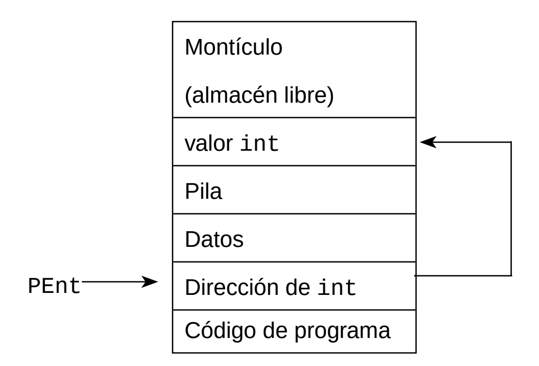


Figura 10.3. Después de new, sobre un entero, pEnt apunta a la posición del montículo donde se ha asignado espacio para el entero.

Ambos ejemplos declaran un puntero denominado BloqueMem y lo inicializan a la dirección devuelta por new. Si un bloque del tamaño solicitado está disponible, new devuelve un puntero al principio de un bloque de memoria del tamaño especificado. Si no hay bastante espacio de almacenamiento dinámico para cumplir la petición, new devuelve cero o NULL.

La reserva de *n* caracteres se puede declarar así:

```
int n;
char *s;
...
cin >> n;
s = new char[n];
```

El operador new está disponible de modo inmediato en C++, de forma que no se requiere ningún archivo de cabecera. El operador devuelve un puntero; en el caso anterior un puntero a un carácter.

Ejemplo 10.1

```
//demonew.cpp

#include <iostream>
#include <string>          // uso de strcpy()
using namespace std;

void main()
{
   char *cad = "Sierras de Cazorla, Segura y Magina";
   int lon = strlen(cad);

   char *ptr;
   ptr = new char[lon+1];

   strcpy(ptr, cad);         // copia cad a nueva área de memoria
                             // apuntada por ptr
   cout << endl << "ptr = " << ptr; //  cad esta ahora en ptr
   delete ptr;               // libera memoria de ptr
}
```

La expresión

```
ptr = new char[lon+1];
```

devuelve un puntero que apunta a una sección de memoria bastante grande para contener la cadena de longitud `strlen()` más un byte extra por el carácter `'\0'` al final de la cadena.

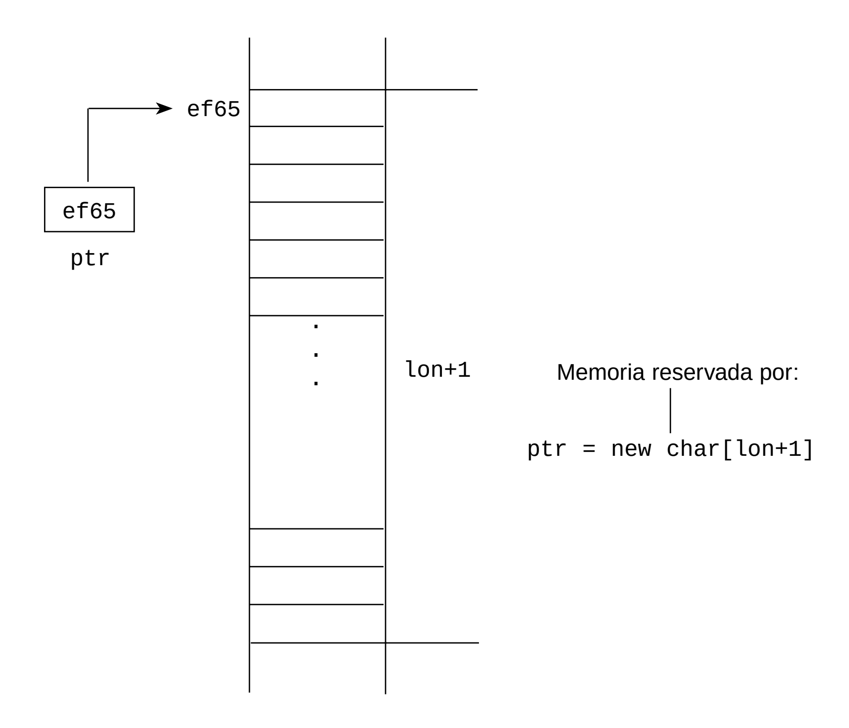


Figura 10.4. Memoria obtenida por el operador new.

Precaución

El almacenamiento libre no es una fuente inagotable de memoria. Si el operador `new` se ejecuta con falta de memoria, se devuelve un puntero `NULL`. Es responsabilidad del programador comprobar *siempre* el puntero para asegurar que es válido antes de que se asigne un valor al puntero. Supongamos, por ejemplo, que se desea asignar un array de 8.000 enteros:

```
int *ptr_lista = new int[8000];
if (ptr_lista == NULL)
{
   cout << "Falta memoria" << endl;
   return -1;    // Hacer alguna acción de recuperación
}

for (int i = 0; i < 8000; i++)
   ptr_lista[i] = lista(i);
```

Existe otro sistema que realiza la misma tarea anterior: utilizar la función **`_set_new_handler()`**.

Si no existe espacio de almacenamiento suficiente, el operador `new` devuelve `NULL`. La escritura de un programa totalmente seguro, exige comprobar el valor devuelto por `new` para asegurar que no es `NULL`. `NULL` es una constante predefinida en C++. Se debe referenciar los archivos de cabecera `#include <iostream>, <stdlib>` y otros para obtener la definición de `NULL`.

Ejemplo 10.2

El programa `TESTMEM` *comprueba aproximadamente cuánta memoria se puede asignar (está disponible).*

```
// TESTMEM: Este programa comprueba cuánta memoria hay
// disponible
#include <iostream>
using namespace std;

void main()
{
   char *p;
   for (int i = 1;  ; i++)
   {
      p = new char[10000];
      if (p == 0) break;
      cout << "Asignada: " << 10 * i << "kB\n";
   }
}
```

Se asigna repetidamente 10 kB (Kilobytes) hasta que falla la asignación de memoria y el bucle se termina.

10.2.1. Asignación de memoria de un tamaño desconocido

Se puede invocar al operador `new` para obtener memoria para un array, incluso si no se conoce con antelación cuánta memoria requieren los elementos del array. Todo lo que se ha de hacer es invocar al operador `new` utilizando un puntero al array. Si new no puede calcular la cantidad de memoria que se necesita cuando se compile su sentencia, `new` espera hasta el momento de la ejecución, cuando se conoce el tamaño del array, y asigna, a continuación, la memoria necesaria.

Por ejemplo, este segmento de código asigna memoria para un array de diez cadenas en el momento de la ejecución:

```
miCadena *miTexto;
miTexto = new miCadena[10];
```

10.2.2. Inicialización de memoria con un valor

Cuando se asigna memoria con el operador `new`, se puede situar un valor o expresión de inicialización dentro de paréntesis al final de la expresión que invoca el operador `new`. C++ inicializa entonces la memoria que se ha asignado al valor que se ha especificado.

Por ejemplo, esta sentencia inicializa y asigna almacenamiento para un entero, inicializa el almacenamiento que se asigna a un valor de 0 y, a continuación, guarda un puntero al bloque de memoria asignado en la variable puntero `memPtr`:

```
int *memPtr = new int(0);
```

Sin embargo, no se puede utilizar la sentencia anterior para inicializar almacenamiento para cada elemento de un array.

```
int *memPtr = new int[100](0);         // No funciona
```

La inicialización no está disponible para los tipos de datos array.

```
p = new int(999);
```

inicializador

equivale a inicializar `*p`

```
*p = 999;
```

10.2.3. Uso de new para arrays multidimensionales

Para asignar un array multidimensional, se indica cada dimensión del array de igual forma que se declara un array multidimensional. Por ejemplo, para asignar un puntero a un array de nombre meses de dimensiones `[12]` por `[10]`, escriba

```
p = new char[12][10]
```

Cuando se utiliza `new` para asignar un array multidimensional, sólo el tamaño más a la izquierda puede ser una variable. Cada una de las otras dimensiones debe ser un valor constante.

10.2.3. Agotamiento de memoria

Puede suceder que una computadora no tenga espacio suficiente de memoria disponible para cumplir una petición de `new`. Cuando sucede esta situación, el operador `new` devuelve el valor 0. En C++, un puntero con valor 0 se denomina *puntero nulo*. La sintaxis de C++ garantiza que el puntero nulo nunca apunte a datos válidos, de tal forma que, con frecuencia, se suele utilizar para indicar fallos de operadores o funciones que, en caso contrario, devuelven punteros útiles.

10.3. EL OPERADOR *delete*

El operador `delete` asegura un uso seguro y eficiente de la memoria; libera la memoria reservada con `new`.

Ejemplo

```
int * ps = new int // asigna memoria con new
...                // uso de la memoria
delete ps          // libera la memoria
```

La acción de `delete` elimina la memoria a la cual apunta `ps` pero no borra el puntero `ps` que se puede seguir utilizando; es decir, se puede utilizar `ps`, por ejemplo, para apuntar a otra asignación de `new`. Es importante que exista siempre una correspondencia `new-delete`.

No se puede utilizar `delete` para liberar memoria creada por declaración de variables ordinarias.

```
int * p = new int; // correcto
delete p;          // correcto
delete p;          // no es correcto, duplicar
int num = 10;      // correcto
int * pn = & num;  // correcto
delete pn;         // incorrecto, memoria no
                   // asignado por new
```

Precaución

No se pueden crear dos punteros al mismo bloque de memoria.

Cuando se ha terminado de utilizar un bloque de memoria previamente asignado por new, se puede liberar el espacio de memoria y dejarlo disponible para otros usos, mediante el operador `delete`. El bloque de memoria suprimido se devuelve al espacio de almacenamiento libre, de modo que habrá más memoria disponible para asignar otros bloques de memoria. El formato es

`delete` *`puntero`*

Así, por ejemplo, para las declaraciones

```
1. int *ad;
     ad = new int       o bien       int *ad = new int

2. char *adc;
     adc = new char[100];
```

el espacio asignado se puede liberar con las sentencias

```
delete ad;
```

o bien

```
delete adc;
```

Ejemplo 10.3

```
gato *pgato = new gato[55]; // Asignar pgato
if (pgato == NULL)
   cout << "Memoria agotada" << endl;
else
{
   ...
   delete pgato;            // Liberar memoria asignada por pgato
}
```

10.4. EJEMPLOS QUE UTILIZAN *NEW* Y *DELETE*

Se puede utilizar el operador `new` con cualquier tipo de dato, incluyendo `char, int, float`, arrays, estructuras e identificadores de `typedef`. El uso de `new` es adecuado para manejo de arrays.

Consideremos una cadena de caracteres formado por un array de caracteres (`char`). Utilizando `new` se puede crear un puntero a un array de longitud variable, de modo que se puede ajustar la cantidad de memoria necesaria para la cadena de texto durante la ejecución del programa. En primer lugar, se realiza la asignación de una cadena de caracteres de 80 bytes. Crear un puntero a `char`, como:

```
char *p;
```

Para asignar la cadena de caracteres se escribe

```
p = new char[80];
```

El sistema de gestión de memoria reserva 80 bytes de memoria para el array de caracteres y la dirección de la asignación se almacena en `p`.

Ejemplo 10.4

```
// ASIGMEM1.CPP
// Muestra la asignación dinámica de una cadena de caracteres

#include <iostream>
#include <string>

void main(void)
{
   char *p;

   p = new char[80];

   strcpy(p, "Carchelejo está en la Sierra Magina")

   cout << p << endl;

   delete p;

   cout << "Pulse Intro para continuar";
   cin.get();
}
```

Una nueva característica de `new` es que no ha de utilizar una constante para el tamaño de la asignación. Utilizando una variable se puede ajustar el tamaño de la dimensión del array a medida que se ejecuta el programa. El programa siguiente muestra cómo asignar una cantidad variable de memoria, a medida que se necesita.

```
// ASIGMEM2.CPP
// Asignación dinámica de una cadena de caracteres

#include <iostream>
#include <string>
```

```
void main(void)
{
   int tama_cadena;
   char *p;

   cout << "¿Cuántos caracteres se asignan?";
   cin. << tama_cadena;
   cin.ignore(1);

   p = new char[tama_cadena];

   strcpy(p, "Carchel también está en Sierra Magina");

   cout << p << endl;

   delete p;
   cout << "Pulse Intro para continuar";
   cin.get();
}
```

Ejercicio 10.1.

Reserva de memoria para tipos reales `(double)` *y al que se le asigna un valor* `4.5`

```
double * p = new double (4.5);
```

Ejemplo

```
//demonew.cpp
//uso del operador new
#include <iostream>
#include <cstring> //función strlen
using namespace std;

int main()
{
   char * cad = "Mi pueblo es Carchelejo en Jaén";
   int long = strlen (cad)      // obtiene longitud de cad

   char * p;                    //puntero a char
   P = new char [long+1];       // memoria a cad + '\0'

   strcpy (p, cad);            // copia cad a p

   cout << "p = " << p << endl; // p está ahora en cad

   delete [] p;
   return 0;
}
```

Ejericio 10.2

Reservar memoria para una estructura; rellenar sus campos, visualizar y liberar la memoria reservada.

```
#include <iostream>
using namespace std;

struct ficha_p {
   int numero;
   char nombre [30];
};

   int main (void)
{
   struct ficha_p  *una_ficha;
   una_ficha = new struct ficha_p;
   cont << "Introduzca el número de cliente: " ;
   cin >> una_ficha.numero;
   cont << "Introduzca un nombre:";
   cin >> una_ficha.nombre –> nombre;
   cont << "\nNumero: " << una_ficha –> numero;
   cont << "\nNombre; " << una_ficha –> nombre;

   delete una_ficha;
}
```

Precaución

Utilizar `delete` sólo para liberar memoria asignada por `new`.

10.5. ASIGNACIÓN DE MEMORIA PARA ARRAYS

La gestión de listas y tablas mediante arrays es una de las operaciones más usuales en cualquier programa. La asignación de memoria para arrays es, en consecuencia, una de las tareas que es preciso conocer en profundidad.

El listado `ASIGMEM3.CPP` muestra cómo se puede utilizar el operador `new` *para* asignar memoria a un array. El programa imprime un mensaje en la salida estándar (normalmente la pantalla).

```
// ASIGMEM3.CPP
// Asignación de memoria para un array

#include <iostream>
using namespace std;

int main()
{
   int *Dias = new int[3];
   Dias[0] = 15;
```

```
   Dias[1] = 8;
   Dias[2] = 1999;
   cout  << "Las fiestas del pueblo son en Agosto "
         << Dias[0] << "/ "
         << Dias[1] << "/ "
         << Dias[2];
   delete [] Dias;        // Libera memoria
   return 0;
}
```

La salida del programa anterior es:

```
Las fiestas del pueblo son en Agosto 15/8/1999
```

10.5.1. Asignación de memoria interactivamente

El programa `ASIGMEM4.CPP` muestra cómo se puede invocar a `new` para asignar memoria para un array. Cuando se ejecuta el programa, se pide al usuario teclear el tamaño de un array. Cuando se contesta adecuadamente el programa imprime su array, que contiene el número de números consecutivos que se han solicitado.

```
// ASIGMEM4.CPP
// Asignación dinámica de arrays

#include <iostream>
using namespace std;

#include <stdlib>

int main()
{
   cout << "¿Cuántas entradas de su array?";
   int lon;
   cin >> lon;
   int *miArray = new int[lon];
   for (int n = 0; n < lon; n++)
      miArray[n] = n + 1;
   for (n = 0; n < lon; n++)
      cout << '\n' << miArray[n];
   delete [] miArray;
   return 0;
}
```

10.5.2. Asignación de memoria para un array de estructuras

El programa `ASIGNAES.CPP` define un modelo de estructura `perro`. A continuación, el programa utiliza el operador `new` para asignar espacio suficiente para contener un array de tres estructuras `animal`. La dirección del bloque de memoria se almacena en el puntero `panimal`. Después de comprobar si tiene éxito la operación de asignación del bloque de memoria, el programa asigna valores a los miembros de cada elemento de la estructura. La función de biblioteca **`strcpy()`** se utiliza para copiar una constante de cadena en los miembros arrays de caracteres `raza` y `color` de la estructura `perro`. Por último, el programa visualiza el contenido de los tres elementos estructura del array `pperro.`

```
// ASIGNAES.CPP
#include <iostream>
using namespace std;

#include <string>

struct perro
{
   char raza[20];
   int edad;
   int altura;
   int color[15];
};

void main()
{
   perro *pperro = new perro[3];
   if (pperro == NULL)
      cout << "**Falta de memoria**" << endl;
   else
   {
      strcpy(pperro[0].raza, "Pastor aleman");
      strcpy(pperro[0].color, "Rubio");
      pperro[0].edad = 4;
      pperro[0].altura = 120;

      strcpy(pperro[1].raza, "dalmata");
      strcpy(pperro[1].color, "blanco y negro");
      pperro[1].edad = 5;
      pperro[1].altura = 130;

      strcpy(pperro[2].raza, "doberman");
      strcpy(pperro[2].color, "negro");
      pperro[2].edad = 4;
      pperro[2].altura = 155;

      for (int i = 0; i < 3; i++)
      {
         cout << "\nRaza:"<< pperro[i].raza << endl;
         cout << "Color:" << pperro[i].color << endl;
         cout << "Altura:"<< pperro[i].altura << endl;
         cout << "Edad:"  << pperro[i].edad << endl;
      }
   }
}
```

Al ejecutarse el programa se visualiza

```
Raza:   pastor aleman
Color:  rubio
Altura: 120
Edad:   4

Raza:   dalmata
```

```
Color:  blanco y negro
Altura: 130
Edad:   5

Raza:   doberman
Color:  negro
Altura: 155
Edad:   4
```

10.6. ARRAYS DINÁMICOS

Normalmente `new` se utiliza para trabajar con bloques de datos, tales como *arrays*, cadenas y estructuras. Ésta es su mejor aplicación. Supongamos que desea escribir un programa que puede necesitar o no, un array, dependiendo de la información que se introduzca al programa durante su ejecución. Si se crea un array declarándolo previamente, se arregla espacio de memoria cuando se compila el programa. La asignación de la memoria al array durante el tiempo de compilación se denomina *ligadura estática*, significando que el array se construye (se crea) durante el tiempo de compilación. Sin embargo, con `new` se puede crear un array durante el tiempo de ejecución si es necesario y si no se omite la creación del array; o incluso se puede seleccionar el tamaño de un array después que el programa se ejecuta. En este caso, la ligadura se llama *ligadura dinámica*, y significa que se crea el array durante la ejecución del programa.

En el caso de la ligadura dinámica, el array toma el nombre de *array dinámico*.

> ### Nota
>
> Durante la ligadura estática, el tamaño del array se debe especificar durante la escritura del programa. Durante la ligadura dinámica, el tamaño del array se puede decidir mientras se ejecuta el programa.

Un nombre de un array es realmente un puntero constante que se asigna en tiempo de compilación:

```
float m[30];// m es un puntero constante a un bloque de 30 float
float* const p = new float[30];
```

`m` y `p` son punteros constantes a bloques de 30 números reales (`float`). La declaración de `m` se denomina *ligadura estática* debido a que se asigna en tiempo de compilación; el símbolo se enlaza a la memoria asignada aunque no se utiliza nunca durante la ejecución del programa.

Por el contrario, se puede utilizar un puntero no constante para posponer la asignación de memoria hasta que el programa se esté ejecutando. Este tipo de enlace o ligadura se denomina *ligadura dinámica* o *ligadura en tiempo de compilación*[1].

```
float* p = new float[30];
```

Un array que se declara de este modo se denomina ***array dinámico***.

Comparar dos métodos de definición de un array

```
• float m[30];                       // array estático
• float * p = new float[30];         // array dinámico
```

[1] Esta propiedad del compilador es la que permite la propiedad de **polimorfismo**.

El *array estático* `m` se crea en tiempo de compilación; su memoria permanece asignada durante toda la ejecución del programa.

El *array dinámico*[2] se crea en tiempo de ejecución; su memoria se asigna sólo cuando se ejecuta su declaración. No obstante, la memoria asignada al array `p` se libera tan pronto como se invoca el operador `delete`, de este modo

```
delete []  p;        // se incluye [] ya que p es un array.
```

Notas sobre la creación de un array dinámico con `new`

Como ya se ha visto, es fácil crear un array dinámico en C++; basta indicar `new`, el tipo de los elementos del array y el número de elementos que desea. Cuando se utiliza `new` para crear un array se debe utilizar `delete` para liberar su memoria.

```
int * pd = new int [100];
...
delete [] pd;      //libera un array dinámico
```

Obsérvese que los corchetes están entre `delete` y el nombre del puntero. Si se utiliza `new` sin corchetes se debe utilizar `delete` sin corchetes. Si se utiliza `new` con corchetes se debe utilizar `delete` con corchetes.

Notas de compatibilidad con versiones antiguas de C++

Las versiones antiguas de C++ pueden no reconocer la notación de los corchetes. Sin embargo, en el caso del estándar ANSI/ISO la discordancia en los paréntesis produce efectos no definidos y un comportamiento no previsto.

Ejemplo

Comportamiento no definido

```
int * ptr1 = new int;
short * ptr2 = new short [100];
delete [] ptr;   //incorrecto
delete ptr 2;    //incorrecto
```

> **Nota**
>
> Recuerde el formato del operador `new`, cuya `innovación` asegura un bloque de memoria suficiente para contener los elementos previstos (*`num_elementos`*) del tipo *`nombre_tipo`* y un *`nombre_puntero`* apuntando al primer elemento
>
> *`nombre_tipo`* *`nombre_puntero`* `= new` *`nombre_tipo`* `[`*`num_elementos`*`];`

[2] En *C++ Primer Plus, five edition*, de Stephen Prata, Sams (2005), se encuentra un amplio y documentado estudio sobre *arrays* dinámicos.

10.6.1. Utilización de un array dinámico

Una vez que se ha creado un array dinámico es necesario conocer cómo utilizar eficientemente dicho array. Así, la sentencia

```
int * p = new int [100];   //crea un bloque de 100 enteros;
```

crea un puntero `p` que apunta al primer elemento de un bloque de `100` valores enteros `(int)`

`p`	*apunta al primer elemento de la array*
`*p`	*es el valor del primer elemento*
`p[0]`	*también equivale al valor del primer elemento del array*
`p[0]` *y* `*p`	*son equivalentes*
`p[1]`	*es el valor del segundo elemento*
`...`	
`p[95]`	*es el valor del elemento número 100*

Ejercicio 10.3

Crear un array dinámico y, a continuación, utilizar la notación de arrays para acceder a sus elementos.

```
//arraydina.cpp
#include <iostream>
using namescape std;

int main()
{
   double *pd = new double [4];   //4 elementos double
   pd[0] = 7.5;
   pd[1] = 4.25;
   pd[2] = 0.45;
   pd[3] = 1.23;
   cunt << " pd[1] es: " << pd[1] << "\n";
   pd = pd + 1;      // incrementar el puntero
   cunt << "ahora pd [1] es" << pd[1];

   delete [] pd;     // liberar la memoria
   return [0];
}
```

Si se ejecuta el programa, la salida es

```
pd[1] es: 4.25
ahora pd [1] es: 0.45
```

10.7. GESTIÓN DEL DESBORDAMIENTO DE MEMORIA: *SET_NEW_HANDLER*

Cuando en un programa se examina el operador `new` y se detecta falta de espacio en memoria, C++ permite definir una función que se puede llamar para manejar los errores. Esta función es **`_set_new_handler(),`** que se define en el archivo de cabecera `new.h`, tiene un puntero a una función como argumen-

to. Cuando se llama a `set_new_handler()`, el puntero que se posiciona se fija para apuntar a una función de manejo de error, que también se proporciona.

Ejemplo 10.5

```
#include <iostream>
using namespace std;

void main()
{
   void desborde();
   set_new_handler (desborde);
   long tamanyo;
   int *dir;
   int nbloque;
   cout << "¿Tamaño (longitud) deseado?";
   cin >> tamanyo;
   for (nbloque = 1; ; nbloque++
   {
      dir = new int[tamanyo];
      cout << "Asignación bloque número:" << nbloque <<
      endl;
   }
}

void desborde()
{
   cout << "Memoria insuficiente-parar ejecución << endl;
   exit(1);
}
```

Si se ejecuta este programa, se producirá una salida tal como:

```
¿Tamaño (longitud) deseado?
Asignación bloque número: 1
Asignación bloque número: 2
Asignación bloque número: 3
Asignación bloque número: 4
Asignación bloque número: 5
Memoria insuficiente - parar ejecución
```

Ejemplo 10.6

El siguiente programa ejecuta un bucle que consume memoria en incrementos de 10.000 bytes, hasta que se agota la memoria. Entonces, se detiene la asignación de memoria y se visualiza la cantidad total de memoria actualmente asignada y se imprime el mensaje `"Almacenamiento libre vacío"`.

```
#include <iostream>
using namespace std;

#include <stdlib>
#include <new.h>

int agotada_mem(size_t tamanyo);
```

```
void main()
{
   set_new_handler(agotada_mem);
   long total = 0;
   while (1)
   {
      char *EspMem = new char[10000];
      total += 10000;
      cout << "Gasto de 10000 bytes" << total << endl;
   }
}

int agotada_mem(size_t tamanyo)
{
   cerr << "\n Almacenamiento libre vacío \n";
   exit(1);
   return 0;
}
```

Obsérvese que en el programa anterior la función **`agotada_mem`** toma un parámetro de tipo `size_t`, que representa el tamaño del bloque solicitado cuando falla `new`.

10.8. REGLAS DE FUNCIONAMIENTO DE *new* Y *delete*

Como ya se ha comentado se puede asignar espacio para cualquier objeto dato de C y C++.

Las reglas para utilizar `new` como medio para obtener espacio libre de memoria son las siguientes:

1. *El operador* `new` *es un operador unitario.*

   ```
   int* datos;
   ...
   datos = new int;
   ```

2. *El operando de new es un nombre de un tipo.*

   ```
   #include <iostream>
   using namespace std;

   void main()
   {
      int* valor;     // Puntero a un entero

      valor = new int;    // Asigna espacio en
                          // almacenamiento libre
      cout << valor << "\r\n";;
      delete valor;       // Destruir enteros
   }
   ```

3. *El operador* `new` *devuelve un puntero del tipo correcto.*
   Si no se encuentra espacio libre disponible, se devuelve un puntero nulo.
4. *El operador* `new` *calcula automáticamente el tamaño del objeto para el que está asignando memoria* `new`*.*
5. *Se puede utilizar* `new` *para asignar espacio para objetos más complejos, tales como arrays, en el almacenamiento libre.*

Para utilizar `new` en asignación de memoria para una cadena o un array, escriba simplemente el nombre del tipo y, a continuación, el tamaño del array encerrado por el declarador del array (`[ ]`). El listado siguiente demuestra este uso de `new`.

```
#include <string>
#include <iostream>
using namespace std;

void main()
{
   char* cad;          // Puntero a una cadena

   cad = new char[41]; // Asignar array
   strcpy(cad, "Un programa de saludo: \r\n");
   cout << cad;
   delete cad;         // Destruir cadena
}
```

6. *Se pueden crear arrays multidimensionales de objetos con* `new`, *pero con algunas restricciones.*

```
print = new int[2][2][2];
```

La primera dimensión puede ser cualquier expresión legítima, tal como:

```
int i = 2;
print = new int[i][2][2];
```

7. *No se pueden crear referencias con* `new`.
8. *Los objetos creados con* `new` *no se inicializan automáticamente a ningún valor.*

```
int* i = new int(35);       // Fija int a 35
```

Las reglas para utilizar el operador `delete` son también sencillas:

1. *Al igual que* `new`, *el operador* `delete` *es un operador unitario.*

```
double *x = new double;
delete x;
...
int* i = new int[256];
delete[] i;
```

2. *Utilizar el operador* `delete` *sólo para almacenamiento libre adquirido por* `new`.
3. *No se puede eliminar un puntero a un constante.*

Reglas para usar arrays dinámicos[3]

Es importante observar las reglas siguientes cuando se utilizan los operadores `new` y `delete`:

- No utilizar `delete` para liberar memoria que no haya sido asignada con `new`.
- No utilizar `delete` para liberar el mismo bloque de memoria dos veces seguidas en sucesión.
- Utilizar `delete[]` si se utiliza `new[]` para asignar un array.
- Utilizar `delete`, sin corchetes, si se utiliza `new` para asignar una entidad simple.
- Es seguro aplicar `delete` al puntero nulo, ya que no sucederá nada.

---

[3] [Prata 2005]. *C++ Primer Plus.* Indianapolis (USA): Sams, 2005, pp. 154-155.

10.9. TIPOS DE MEMORIA EN C++

Existen tres tipos de memoria en C++ (almacenamiento de datos): *memoria o almacenamiento automático, memoria o almacenamiento estático y memoria o almacenamiento dinámico*.

Memoria automática (almacenamiento automático)

Las variables definidas en el interior de una función se denominan *variables automáticas* y utilizan *almacenamiento automático.* Esto significa que las variables se crean y reservan espacio automáticamente cuando se invocan y se agotan (desaparece la reserva de memoria) cuando termina la función.

Los valores automáticos son locales al bloque los contiene. Un *bloque* es una sección de código encerrado entre llaves; en el caso más usual: funciones.

```
int main()
{
   ...
}

char * leernombre()
{
   char tabla[100];   //almacenamiento temporal
   ...
}
```

Memoria estática (almacenamiento estático)

El almacenamiento estático se reserva durante la ejecución de un programa completo. Existen dos métodos para hacer una variable estática:

1. Definir la variable externamente fuera de una función.
2. Utilizar la palabra reservada `static` cuando se declara la variable

```
static double temperatura = 25.75;
```

- *En C se pueden inicializar sólo arrays y estructuras estáticas.*
- *En ANSI/ISO C++ y en ANSI C se pueden inicializar arrays y estructuras dinámicas.*

Memoria dinámica (almacenamiento dinámico)

Los operadores `new` y `delete` gestionan memoria libre que es independiente de la memoria utilizada en variables estáticas y automáticas. El uso de `new` y `delete` juntas proporcionan más control eficiente sobre el uso de la memoria que en el caso de usar variables ordinarias.

La memoria dinámica es un área de memoria disponible y que el programador puede asignar (reservar) o liberar. En ausencia de un *recolector automático de basura («garbage collection»),* como sucede en el lenguaje Java, la memoria permanece ocupada (reservada) hasta que el programador, explícitamente, libera dicha memoria devolviendo su espacio ocupado a la zona de memoria dinámica libre, momento a partir del cual puede ser utilizada de nuevo.

10.9.1. Problemas en la asignación dinámica de memoria

Cuando se utiliza el operador `new`, el programa realiza una petición al sistema operativo, solicitando memoria. El sistema responde comprobando la memoria disponible y viendo si hay espacio disponible.

En las computadoras actuales existen grandes cantidades de memoria disponibles[4]. A no ser que se soliciten espacios excepcionales de memoria, se debe esperar casi siempre poder disponer de la memoria solicitada. Pero es posible que la memoria pueda no estar disponible y, por consiguiente, se debe tener siempre presente esta posibilidad.

En el caso de que no exista disponible la memoria solicitada, el operador `new` devuelve un puntero nulo. Se puede comprobar esta posibilidad y realizar, en su caso, la acción oportuna

```
int *p = new int[5000];
if (!p) {
   cont << "Memoria insuficiente";
   exit (0);
}
```

Otro problema que puede ocurrir se debe a la *«pérdida de memoria»* (*memory leaks*). Cuando se solicita memoria con `new`, el sistema operativo reserva un bloque de memoria hasta que se libera con `delete`. En el caso de que se termine el programa sin liberar toda la memoria asignada dinámicamente, el sistema va perdiendo memoria cada vez que se ejecuta el programa, de modo que su computadora tendrá cada vez menos memoria disponible.

Para evitar el problema anterior se debe asegurar el uso de `delete` para liberar cualquier memoria asignada dinámicamente antes de que se termine el programa.

Existen lenguajes de programación, como ya se ha comentado anteriormente, que tienen un mecanismo denominado *«recolector de basura»* (*garbage collector*) que cuando se ejecuta el programa, en el caso de encontrar bloques de memoria que no está en uso los borra antomáticamente y libera dichos bloques. **Visual Basic** y **Java** son dos lenguajes que tienen esta característica. C++, por el contrario, no tiene esta propiedad.

Ejercicio 10.4

Aplicación práctica de uso de una memoria dinámica; Asignación dinámica de un array con el tamaño solicitado por el usuario; Almacenamiento de valores, en el array, introducidos por el usuario; Visualización de los valores del array y cálculo de su media aritmética.

```
#include <iostream>
using namespace std;

int main()
{
   int suma = 0;
   int n;
   int *p;

   cunt << "Introduzca número de elementos: ";
   cin >> n;

   p = new int [n];//asignación de n enteros
```

[4] Hoy son ya usuales memorias principales (RAM) de 512 KB y 1 GB, y se comercializan también computadoras personales de 2 y 4 GB, que se encuentran fácilmente en grandes almacenes.

```
        for (int i=0; i<n; 1++)
        {
           cont << "Introduzca elemento: " << i ;
           cin >> p[i];
           suma += p[i];
        }

        cunt << "elementos introducidos: ";
        for (int i = 0; i < n; i++)
           cunt << p[i] << " , " ;
           cunt << endl;
           cunt << "Total: " << Suma << endl;
           cunt << "Media: " << (double) suma /n << endl;

           delete [] p; //Liberación espacio de n enteros

           return ;
     }
```

10.10. ASIGNACIÓN Y LIBERACIÓN DE MEMORIA EN C

En el lenguaje C no existe ningún operador que reserve y libere memoria dinámica al estilo de C++. En su lugar, la manipulación se efectúa a través de las funciones de la biblioteca (`stdlib.h`) declarada en la cabecera: `malloc()`, `calloc()`, `realloc()` y `free()`.

> ***Regla práctica***
>
> C gestiona la memoria dinámica mediante las funciones `malloc()`, `calloc()`, `realloc()` y `free()` de la biblioteca *(stdlib.h)*.

RESUMEN

La asignación dinámica de memoria permite utilizar tanta memoria como se necesite. Se puede asignar espacio a una variable en el almacenamiento libre cuando se necesite y se libera la memoria cuando se desee.

En C se utilizan las funciones **`malloc()`** y **`free()`** para asignar y liberar memoria. Los programadores de Turbo C++ no utilizan **`malloc()`** y **`free()`**, sino dos nuevos operadores, `new` y `delete`, que sustituyen a **`malloc()`** y **`free()`**, respectivamente.

El operador `new` asigna un bloque de memoria que es el tamaño del tipo de objeto especificado (un `int`, un `float`, una estructura, un array o cualquier otro tipo de dato).

Cuando se termina de utilizar un bloque de memoria, se puede liberar con el operador `delete`. La memoria libre se devuelve al almacenamiento libre, de modo que quedará más memoria disponible para asignar otros bloques de memoria.

El siguiente ejemplo asigna un array y llama al operador `delete` que libera el espacio ocupado en memoria:

```
perro* pperro = new perro[3];
if (pperro == NULL)
   cout << "¡Falta memoria! \n";
else
{
   .  // uso de pperro
   .
   .
   delete pperro; // Libera espacio
                  // de perro
}
```

C++ ofrece en su lenguaje una función para manejo de errores producidos por desbordamiento de memoria: **`set_new_handler`**.

EJERCICIOS

10.1. Suponga que un programa contiene el siguiente código para crear un array dinámico:

```
int *entrada-;
entrada = new int[10]-;
```

Si no existe suficiente memoria en el montículo, la llamada anterior a `new` fallará. Escribir un código que verifique si esta llamada a `new` falla por falta de almacenamiento suficiente en el montículo (heap). Visualice un mensaje de error en pantalla que lo advierta expresamente.

10.2. Suponga que su programa contiene código para crear un array dinámico como en el Ejercicio 10.3 de modo que la variable `entrada` está apuntando a este array dinámico. Escriba el código que rellene este array con 10 números escritos por teclado.

10.3. Escribir una función que calcule los elementos de un array de enteros `a,` mediante punteros.

PROBLEMAS

10.1. Escribir una función que utilice punteros para copiar un array de elementos `double`.

10.2. Escribir un programa que lea un número determinado de enteros (1.000, como máximo) del terminal y los imprima en el mismo orden y con la condición de que cada entero sólo se escribe una vez. Si el entero ya se ha impreso, no se debe imprimir de nuevo. Por ejemplo, si los números siguientes se leen desde el terminal:

```
55 78 -25 3 55   24  -3  7
```

el programa debe imprimir lo siguiente:

```
55  78  -25 3 24  -3  7
```

Se deben utilizar punteros para tratar con el array en el que se hayan almacenado los números.

10.3. Escribir el código de la función siguiente que devuelve la suma de elementos `float` apuntados por los `n` primeros punteros del array `p`:

```
float suma (float *p[ ], int n)
```

10.4. Escribir la siguiente función que ordena indirectamente los elementos `float` apuntados por los `n` primeros punteros del array `p` por reordenación de los punteros:

```
void sort(float *p[ ], int n)
```

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 237).
2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

10.1. ¿Cuál es la salida del siguiente programa?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int * p1, *p2;
    p1= new int;
    *p1 = 45;
    p2 = p1;
    cout << " *p1 == " << *p1 << endl;
    cout << " *p2 == " << *p2 << endl;
    *p2 = 54;
    cout << " *p1 == " << *p1 << endl;
    cout << " *p2 == " << *p2 << endl;
```

```
    cout << " vivan los punteros \n";
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

10.2. ¿Cuál es la salida del siguiente código?

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    int * p1, *p2;
    p1= new int;
    p2= new int;
    *p1 = 10;
    *p2 = 20;
    cout <<  *p1 << " " << *p2 << endl;
    *p1 = *p2;
    cout <<  *p1 << " " << *p2 << endl;
    *p1 = 30;
    cout <<  *p1 << " " << *p2 << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

10.3. Suponga que un programa contiene el siguiente código para crear un array dinámico.

```
int * entrada;
entrada = new int[10];
```

Si no existe suficiente memoria en el montículo, la llamada anterior a `new` fallará. Escribir un código que verifique si esa llamada a `new` falla por falta de almacenamiento suficiente en el montículo (heap). Visualice un mensaje de error en pantalla que lo advierta expresamente.

10.4. Suponga que un programa contiene código para crear un array dinámico como en el Ejercicio 11.3, de modo que la variable entrada está apuntado a este array dinámico. Escribir el código que rellene este array de 10 números escritos por teclado.

10.5. Dada la siguiente declaración, definir un puntero `b` a la estructura, reservar memoria dinámicamente para una estructura asignando su dirección a `b`.

```
struct boton
{
    char* rotulo;
    int codigo;
};
```

10.6. Una vez asignada memoria al puntero `b` del Ejercicio 10.5 escribir sentencias para leer los campos `rotulo` y `codigo`.

10.7. Declarar una estructura para representar un `punto` en el espacio tridimensional con un nombre. Declarar un puntero a la estructura que tenga la dirección de un array dinámico de n estructuras `punto`. Asignar memoria al array y comprobar que se ha podido asignar la memoria requerida.

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 239).

2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

10.1. Dada la declaración de la estructura punto del Ejercicio 11.7, escribir una función que devuelva la dirección de un array dinámico de n puntos en el espacio tridimensional. Los valores de los datos se leen del dispositivo de entrada (teclado).

10.2. Añadir al problema anterior una función que muestre los puntos de un vector dinámico cuya tercera coordenada sea mayor que un parámetro que recibe como dato.

10.3. Añadir al problema anterior el punto más alejado del origen de coordenadas.

10.4. Se pretende representar en el espacio tridimensional triángulos, pero minimizando el espacio necesario para almacenarlos. Teniendo en cuenta que un triángulo en el espacio tridimensional viene determinado por las coordenadas de sus tres vértices, escribir una estructura para representar un triángulo que usando el vector

dinámico de los problemas anteriores, almacene el índice de los tres vértices así como su perímetro y área (de esta forma se almacena los tres vértices con tres enteros en lugar de con 9 reales, ahorrando memoria). El perímetro y el área deben ser calculados mediante una función a partir del índice de los tres vértices, y el vector dinámico de puntos del espacio tridimensional.

10.5. Añadir al problema anterior una función que lea y almacene en un vector los m triángulos del espacio tridimensional cuyos vértices se encuentren en el array dinámico de puntos. Usar las funciones y estructuras definidas en los problemas anteriores.

10.6. Añadir al problema anterior una función que visualice los triángulos del vector de triángulos.

10.7. Dada la declaración del array de punteros:

```
#define N 4
char *linea[N];
```

Escribir las sentencias de código para leer N líneas de caracteres y asignar cada línea a un elemento del array.

10.8. Escribir un código de programa que elimine del array de N líneas del Problema 11.7 todas aquellas líneas de longitud menor que 20.

10.9. Escribir una función que utilice punteros para copiar un array de elementos `double`.

10.10. Escribir un programa que lea un número determinado de enteros (1000 como máximo) del terminal y los visualice en el mismo orden y con la condición de que cada entero sólo se escriba una vez. Si el entero ya se ha impreso, no se debe visualizar de nuevo. Por ejemplo, si los números siguientes se leen desde el terminal `55 78 –25 55 24 –3 7` el programa debe visualizar lo siguiente: `55 78 –25 55 24 –3 7`. Se debe tratar con punteros para tratar con el array en el que se hayan almacenado los números.

10.11. Escribir un programa para leer n cadenas de caracteres. Cada cadena tiene una longitud variable y está formada por cualquier carácter. La memoria que ocupa cada cadena se ha de ajustar al tamaño que tiene. Una vez leídas las cadenas se debe realizar un proceso que consiste en eliminar todos los blancos, siempre manteniendo el espacio ocupado ajustado al número de caracteres. El programa debe mostrar las cadenas leídas y las cadenas transformadas.

10.12. Escribir el código de la siguiente función que devuelve la suma de los elementos `float` apuntados por los `n` primeros punteros del array `p`.

10.13. Escribir una matriz que reciba como parámetro un puntero doble a `float`, así como su dimensión, y retorne memoria dinámica para una matriz cuadrada.

10.14. Escribir una función que genere aleatoriamente una matriz cuadrada simétrica, y que pueda ser llamada desde un programa principal que genere aleatoriamente la matriz con la función del Problema 11.13. El tratamiento de la matriz debe hacerse con punteros.

10.15. Escribir una función que decida si una matriz cuadrada de orden `n` es mayoritaria. “una matriz cuadrada de orden `n` se dice que es mayoritaria, si existe un elemento en la matriz cuyo número de repeticiones sobrepasa a la mitad de `n*n`. El tratamiento debe hacerse con punteros.

10.16. Una malla de números enteros representa imágenes, cada entero expresa la intensidad luminosa de un punto. Un punto es un elemento “ruido” cuando su valor se diferencia en dos o más unidades del valor medio de los ocho puntos que le rodean. Escribir un programa que tenga como entrada las dimensiones de la malla, reserve memoria dinámicamente para una matriz en la que se lean los valores de la malla. Diseñar una función que reciba una malla, devuelva otra malla de las mismas dimensiones donde los elementos “ruido” tengan el valor 1 y los que no lo son valor 0.

CAPÍTULO 11
Cadenas

Contenido

INTRODUCCIÓN

C++ (y su lenguaje padre, C) no tiene datos predefinidostipo cadena (*string*). En su lugar, C++, como C, manipula cadenas mediante arrays de caracteres que terminan con el carácter nulo ASCII ('\0'). Una *cadena* se considera como un array unidimensional de tipo char o unsigned char. En este capítulo se estudiarán temas tales como:

- cadenas en C++;
- lectura y salida de cadenas;
- uso de funciones de cadena de la biblioteca estándar;
- asignación de cadenas;
- operaciones diversas de cadena (longitud, concatenación, comparación y conversión);
- localización de caracteres y subcadenas;
- inversión de los caracteres de una cadena.

CONCEPTOS CLAVE

- Asignación.
- Cadena.
- Cadena vacía.
- Carácter nulo (`NULL, '\0'`).
- Comparación.
- Conversión.
- Biblioteca *string.h*.
- Inversión.
- Funciones de cadena.
- *String*.

11.1. CONCEPTO DE CADENA

Una *cadena* es un tipo de dato compuesto, un array de caracteres (`char`), terminado por un carácter *nulo* (`'\0'`), `NULL` (Fig. 11.1).

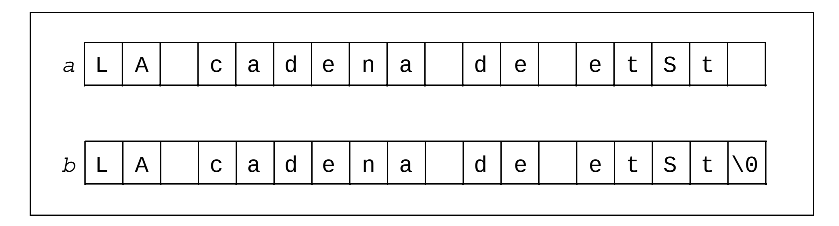


Figura 11.1. *a)* array de caracteres; *b)* cadena de caracteres.

Una cadena (también llamada *constante de cadena* o *literal de cadena*) es `"ABC"`. Cuando la cadena aparece dentro de un programa se verá como si se almacenaran cuatro elementos: `'A', 'B', 'C' y '\0'`. En consecuencia, se considerará que la cadena `"ABC"` es un array de cuatro elementos de tipo char. El valor real de esta cadena es la dirección de su primer carácter y su tipo es un puntero a `char`. Aplicando el operador `*` a un objeto de tipo se obtiene el carácter que forma su contenido; es posible también utilizar aritmética de direcciones con cadenas:

```
*"ABC"            es igual a        'A'
*("ABC" + 1)      es igual a        'B'
*("ABC" + 2)      es igual a        'C'
*("ABC" + 3)      es igual a        '\0'
```

De igual forma, utilizando el subíndice del array se puede escribir:

```
"ABC"[0]      es igual a      'A'
"ABC"[1]      es igual a      'B'
"ABC"[2]      es igual a      'C'
"ABC"[3]      es igual a      '\0'
```

El número total de caracteres de un array es siempre igual a la longitud de la cadena más 1.

Ejemplos

1. `char cad[] = "Mackoy";`
   `cad` tiene siete caracteres: `'M', 'a', 'c', 'k', 'o', 'y' y '\0'`

2. `cout << cad;`

   el sistema copiará caracteres de `cad` a `cout` hasta que el carácter `NULL`, `'\0'` se encuentre.

3. `cin >> bufer;`

   el sistema copiará caracteres de `cin` a `bufer` hasta que se encuentre un carácter espacio en blanco. El usuario ha de asegurarse que el bufer se define como una cadena de caracteres lo suficiente grande para contener la entrada.

Las funciones declaradas en el archivo de cabecera `<string>` en ANSI/ISO C++ y `<string.h>`, en resto de compiladores se utilizan para manipular cadenas.

11.1.1. Declaración de variables de cadena

Las cadenas se declaran como los restantes tipos de arrays. El operador postfijo `[]` contiene el tamaño máximo del objeto. El tipo base, naturalmente, es `char`, o bien `unsigned char`:

```
char texto[80];      // una línea de caracteres de texto
char orden[40];      // cadena utilizada para recibir una orden del
                     // teclado
unsigned char datos; // activación de bit de orden alto
```

El tipo `unsigned char` puede ser de interés en aquellos casos en que los caracteres especiales presentes puedan tener el bit de orden alto activado. Si el carácter se considera con signo, el bit de mayor peso (orden alto) se interpreta como *bit de signo* y se puede propagar a la posición de mayor orden (peso) del nuevo tipo.

Observe que el tamaño de la cadena ha de incluir el carácter `'\0'`. En consecuencia, para definir un array de caracteres que contenga la cadena `"ABCDEF"`, escriba

```
char UnaCadena[7];
```

A veces, se puede encontrar una declaración como ésta:

```
char *s;
```

¿Es realmente una cadena `s`? No, no es. Es un puntero a un carácter (el primer carácter de una cadena).

11.1.2. Inicialización de variables de cadena

Todos los tipos de arrays requieren una inicialización que consiste en una lista de valores separados por comas y encerrados entre llaves.

```
char texto[81] = "Esto es una cadena";
char textodemo[255] = "Ésta es una cadena muy larga";
char cadenatest[] = "¿Cuál es la longitud de esta cadena?";
```

Las cadenas `texto` y `textodemo` pueden contener `80` y `254` caracteres, respectivamente, más el carácter nulo. La tercera cadena, `cadenatest`, se declara con una especificación de tipo incompleta y se completa sólo con el inicializador. Dado que en el literal hay `36` caracteres y el compilador añade el carácter `'\0'`, un total de `37` caracteres se asignarán a `cadenatest`.

Ahora bien, una cadena no se puede inicializar fuera de la declaración. Por ejemplo, si trata de hacer

```
UnaCadena = "ABC";
```

C++ le dará un error al compilar. La razón es que un identificador de cadena, como cualquier identificador de array se trata como un valor de dirección. *¿Cómo se puede inicializar una cadena fuera de la declaración?* Más adelante se verá, pero podemos indicar que será necesario utilizar una función de cadena denominada `strcpy()`.

Ejemplo 11.1

Las cadenas terminan con el carácter nulo. Así, en el siguiente programa se muestra que el carácter `NULL ('\0')` *se añade a la cadena:*

```
main()
{
   char S[] = "ABCD";
   for (int i = 0; i < 5; i++)
      cout << "S[" << i << "] = “ << S[i] << " '\n";
}
```

Ejecución

```
S[0] = 'A'
S[1] = 'B'
S[2] = 'C'
S[3] = 'D'
S[4] = "
```

Comentario

Cuando el carácter `NULL` se envía a `cout`, no se imprime nada.

11.2. LECTURA DE CADENAS

La lectura usual de datos con el objeto `cin` y el operador >>, cuando se aplica a datos de cadena producirá normalmente anomalías. Así, por ejemplo, trate de ejecutar el siguiente programa:

```
// Este programa muestra cómo cin lee datos de cadena
// utilizando el operador cin

#include <iostream>
using namespace std;

void main()
{
   char Nombre[30];            // Definir array de caracteres

   cin >> Nombre;              // Leer la cadena Nombre
   cout << '\n' << Nombre;     // Escribir la cadena Nombre
}
```

El programa define `Nombre` como un array de caracteres de 30 elementos. Suponga que introduce la entrada `Pepe Mackoy`, cuando ejecuta el programa se visualizará en pantalla `Pepe`. Es decir, la palabra `Mackoy` no se ha asignado a la variable cadena `Nombre`. La razón es que el objeto `cin` termina la operación de lectura siempre que se encuentra un espacio en blanco.

Así pues, ¿cuál será el método correcto para lectura de cadenas, cuando estas cadenas contienen más de una palabra (caso muy usual)? El método recomendado será utilizar una función denominada `getline()`, en unión con `cin`, en lugar del operador >>. La función `getline` permitirá a `cin` leer la cadena completa, incluyendo cualquier espacio en blanco.

Ejemplo 11.2

Entrada y salida de cadenas. Lectura de palabras en una memoria intermedia (bufer) de 79 caracteres.

```
main()
{
   char palabra[80];
   do {
      cin >> palabra;
      if (*palabra) cout << "\t\" " << palabra << "\"\n";
   } while (*palabra);
}
```

Al ejecutar este programa el número de veces que se repite el bucle `while` dependerá del número de palabras introducidas, incluido el carácter de control que termina el bucle (`Control-D), ˆD`).

Ejecución

```
Hoy es 1 de Mayo de 1999.
            "Hoy"
            "es"
            "1"
            "de"
            "Mayo"
            "de"
            "1999"
Mañana es Domingo.
            "Mañana"
            "es"
            "Domingo"
ˆD
```

El bucle anterior se ejecuta once veces, una vez por cada palabra introducida (incluyendo **`Control-D`**, que detiene el bucle). Cada palabra del flujo de entrada `cin` hace eco en el flujo de salida `cout`. El flujo de salida no «se limpia» hasta que el flujo de entrada encuentra el final de la línea.

Cada cadena se imprime encerrada entre comillas. Será distinto de cero («*true*») mientras que la cadena `palabra` contiene una cadena de longitud mayor que 0. La cadena de longitud 0 se denomina *cadena vacía*, debido a que contiene el carácter `'\0' (NUL)` en su primer elemento. Pulsando **`Control-D`** (en sistema UNIX o en Linux) o **`Control-Z`** (en Windows/DOS) se envía el carácter final de archivo desde `cin`. Esta acción carga la cadena vacía en `palabra`, fijando `*palabra` (idéntica a `palabra[o]`) a `\0'` y deteniendo el bucle. La última línea de salida muestra sólo el eco de **`Control-D`**.

> ***Advertencia***
>
> Los signos de puntuación (apóstrofes, comas, puntos, etc.) se incluyen en las cadenas, pero no así los caracteres espacios en blanco (blancos, tabulaciones, nuevas líneas, etc.).

11.2.1. Funciones miembro `cin`

El objeto `cin` del flujo de entrada incluye las funciones de entrada: `cin.getline()`, `cin.get()`, `cin.ignore()`, `cin.putback()` y `cin.peek()`. Cada uno de estos nombres de funciones incluyen el prefijo `"cin"`.

`cin` es un objeto de la clase `istream` y **`getline()`** es una función miembro de la clase `iostream`; en consecuencia, `cin` puede llamar a **`getline()`** para leer una línea completa incluyendo cualquier espacio en blanco. La sintaxis de la función **`getline()`** es:

```
istream& getline(signed char* buffer, int long, char separador = '\n');
```

La lectura de cadenas con `cin` se realiza con el siguiente formato:

```
cin.getline(<var_cad>, <max_long_cadena+2>, <'separador'>);
```

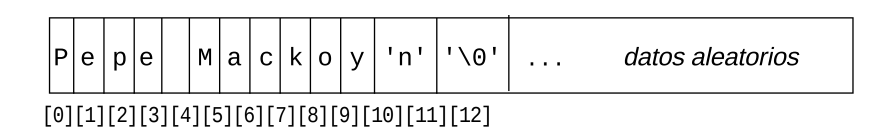
La función **`getline()`** utiliza tres argumentos. El primer argumento es el identificador de la variable cadena (nombre de la cadena). El segundo argumento es la longitud máxima de cadena (el número máximo de caracteres que se leerán), que debe ser al menos dos caracteres mayor que la cadena real, para permitir el carácter nulo `'\0'` y el `'\n'`. Por último, el carácter separador se lee y almacena como el siguiente al último carácter de la cadena. La función **`getline()`** inserta automáticamente el carácter nulo como el último carácter de la cadena.

El ejemplo anterior para leer la cadena `Nombre` se convierte en:

```
#include <iostream>
using namespace std;

void main()
{
   char Nombre[32];           // Define el array de caracteres
   cin.getline(Nombre, 32);  // Lee la cadena Nombre
   cout << Nombre;           // Visualiza la cadena Nombre
}
```

Si se introduce la cadena `Pepe Mackoy`, el array `Nombre` almacenará los caracteres siguientes:



Ejemplo 11.3

```
int #include <iostream>
using namespace std;

main()
{
   char nombre[80];
   cout << "Introduzca su nombre";
   cin.getline(nombre, sizeof(nombre)-2);
   cout << "Hola" << nombre << "¿cómo está usted?";
   return 0;
}
```

El siguiente programa lee y escribe el nombre, dirección y teléfono de un usuario.

```
#include <iostream>
using namespace std;
```

```
void main()
{
   char Nombre[32];
   char Calle[32];
   char Ciudad[27];
   char Provincia[27];
   char CodigoPostal[5];
   char Telefono[10];

   // Leer cadenas nombre y dirección

   cin.getline(Nombre, 32);
   cin.getline(Calle, 32);
   cin.getline(Ciudad, 27);
   cin.getline(Provincia, 27);
   cin.getline(CodigoPostal, 5);
   cin.getline(Telefono, 10);

   // Visualizar cadenas nombre y dirección

   cout << Nombre;
   cout << Calle;
   cout << Ciudad;
   cout << Provincia;
   cout << CodigoPostal;
   cout << Telefono;
}
```

Reglas

- La llamada `cin.getline(cad, n, car)` lee todas las entradas hasta la primera ocurrencia del carácter separador `car` en `cad`.
- Si el carácter especificado `car` es el carácter de nueva línea `'\n'`, la llamada anterior es equivalente a `cin.getline(cad, n)`.

11.2.2. Función `cin.get()`

La función `cin.get()` se utiliza para leer carácter a carácter. La llamada `cin.get(car)` copia el carácter siguiente del flujo de entrada `cin` en la variable `car` y devuelve `1`, a menos que se detecte el final del archivo, en cuyo caso se devuelve `0`.

Ejemplo 11.4

El siguiente programa cuenta las ocurrencias de la letra `'t'` *en el flujo de entrada.*

Un bucle `while` se continúa ejecutando mientras que la función `cin.get (car)` lee caracteres y los almacena en `car`.

```
main()
{
   char car;
```

```
    int cuenta = 0;
    while (cin.get(car))
       if (car == 't') ++cuenta;
    cout << cuenta << "letras t\n";
}
```

Nota

La salida del bucle es `^D`.

11.2.3. Función `cout.put()`

La función opuesta de `get` es `put`. La función `cout.put()` se utiliza para escribir en el flujo de salida `cout` carácter a carácter.

Ejercicio 11.1

El siguiente programa hace «eco» del flujo de entrada y convierte las palabras en palabras iguales que comienzan con letra mayúscula. Es decir, si la entrada es `"sierra de cazorla"` se ha de convertir en `"Sierra De Cazorla"`. Para realizar esa operación se recurre a la función `toupper(car)` que devuelve el equivalente mayúscula de `car` si `car` es una letra minúscula.

```
#include <iostream>
using namespace std;

#include <ctype>
void main()
{
   char car, pre = '\0';
   while (cin.get(car)) {
      if (pre == ' ' || pre == '\n')
         cout.put(char(toupper(car)));
         else cout.put(car);
         pre = car;
   }
}
```

Ejecución

```
sierra de cazorla con capital en Cazorla
Sierra De Cazorla Con Capital En Cazorla
^Z
```

Análisis

La variable `pre` contiene el carácter leído anteriormente. El algoritmo se basa en que si `pre` es un blanco o el carácter nueva línea, entonces el carácter siguiente `car` será el primer carácter de la siguiente palabra. En consecuencia, `car`, se reemplaza por su carácter mayúscula equivalente: `car + 'A' - 'a '`.

11.2.4. Funciones `cin.putback()` y `cin.ignore()`

La función `cin.putback()` restaura el último carácter leído por `cin.get()` de nuevo al flujo de entrada `cin`. La función `cin.ignore()` lee uno o más caracteres del flujo de entrada `cin` sin procesar.

Ejercicio 11.2

El programa siguiente comprueba una función que extrae los enteros del flujo de entrada.

```
int siguienteEntero();

void main()
{
   int m = siguienteEntero(), n = siguienteEntero();
   cin.ignore(80,'\n');
   cout << m << " + " << n << " = " << m + n << endl;
}

int siguienteEntero
{
   char car;
   int n;
   while (cin.get(car))
      if (car >= '0' && car <= '9') {
         cin.putback(car);
         cin >> n;
         break;
      }
   return n;
}
```

Ejecución

```
¿Resultado de 406 más 3121?
406 + 3121 = 3527
```

Análisis

La función `siguienteEntero()` explora los caracteres pasados en `cin` hasta que se encuentra el primer dígito (4 en el ejemplo).

11.2.5. Función `cin.peek()`

La función `cin.peek()` se puede utilizar en lugar de la combinación `cin.get()` y `cin.putback()`. La llamada

```
car = cin.peek()
```

copia el siguiente carácter del flujo de entrada `cin` en la variable `car` de tipo `char` sin eliminar el carácter del flujo de entrada. La función `peek()` se puede utilizar en lugar de las funciones `get()` y `putback`.

Ejemplo 11.5

```
int siguienteEntero()
{
```

```
    char car;
    int n;
    while (car = cin.peek())
       if (car >= '0' && car <= '9')
       {
          cin >> n;
          break;
       }
       else   cin.get(car);
    return n;
}
```

Nota

La expresión `car = cin.peek()` copia el siguiente carácter en `car`. A continuación, si `car` es un dígito, el entero completo se lee en `n` y se devuelve. En caso contrario, el carácter se elimina de `cin` y continúa el bucle. Si se encuentra el final del archivo, la expresión `car = cin.peek()` devuelve 0, y se detiene el bucle.

11.3. LA BIBLIOTECA *string*|*string.h*|*cstring*[1]

La biblioteca estándar de C++ contiene la biblioteca de cadena `string.h`, que contienen las funciones de manipulación de cadenas utilizadas más frecuentemente. Los archivos de cabecera `studio.h`, e `iostream.h` y `iostream` también soportan E/S de cadenas. Algunos fabricantes de C++ también incorporan otras bibliotecas para manipular cadenas, pero como *no son estándar* no se considerarán en esta sección.

El uso de las funciones de cadena tienen una variable parámetro declarada similar a:

```
char *s1
```

Esto significa que la función espera un puntero a una cadena. Cuando se utiliza la función, se puede usar un puntero a una cadena o se puede especificar el nombre de una variable array char. Cuando se pasa un array a una función, C++ pasa automáticamente la dirección del array `char`. La Tabla 11.1 resume algunas de las funciones de cadena más usuales.

11.3.1. La palabra reservada `const`

Las funciones de cadena declaradas en `<string.h>` (en <string> en ANSI/ISO C++) recogidas en la Tabla 11.1 y algunas otras, incluyen la palabra reservada `const`. La ventaja de esta palabra reservada es que se puede ver rápidamente la diferencia entre los parámetros de entrada y salida. Por ejemplo, el segundo parámetro *fuente* de `strcpy` representa el área fuente; se utiliza sólo para copiar caracteres de ella, de modo que este área no se modificará. La palabra reservada **`const`** se utiliza para esta tarea. Se considera un parámetro de *entrada*, ya que la función *recibe* datos a través de ella. En contraste, el primer parámetro *dest* de `strcmpy` es el área de destino, la cual se sobreescribirá, y por consiguiente, no se debe utilizar **`const`** para ello. En este caso, el parámetro correspondiente se denomina *parámetro de salida*, ya que los datos se escriben en el área de destino.

11.4. ARRAYS Y CADENAS COMO PARÁMETROS DE FUNCIONES

Los arrays y cadenas se pueden pasar sólo *por referencia*, no por valor. En la función, las referencias a los elementos individuales se hacen por indirección de la dirección del objeto.

---

[1] Las funciones para manipulación de cadenas estilo C se encuentran en `<string.h>` y en `<cstring>`. Otras funciones de cadena se encuentran en el archivo de cabecera `<stdlib.h>`.

Tabla 11.1. Funciones de <string.h>.

Función	Cabecera de la función y prototipo
`memcpy()`	`void* memcpy(void* s1, const void* s2, size_t n);` Reemplaza los primeros *n* bytes de `*s1` con los primeros *n* bytes de `*s2`. Devuelve `s`.
`strcat`	`char *strcat(char *destino, const char *fuente);` Añade la cadena *`fuente`* al final de *`destino`*.
`strchr()`	`char* strchr(char* s1, const char* s2);` Devuelve un puntero a la primera ocurrencia de `c` en `s`. Devuelve `NULL` si `c` no está en `s`.
`strcmp`	`int strcmp(const char *s1, const char *s2);` Compara la cadena *`s1`* a *`s2`* y devuelve: `0  si s1 = s2` `<0 si s1 < s2` `>0 si s1 > s2`
`strcmpi`	`int strcmpi(const char *s1, const char *s2);` Igual que `strcmp()`, pero trata los caracteres como si fueran todos del mismo tamaño.
`strcpy`	`char *strcpy(char *dest, const char *fuente);` Copia la cadena *`fuente`* a la cadena *`destino`*.
`strcspn()`	`size_t strcspn(char* s1, const char* s2);` Devuelve la longitud de la subcadena más larga de `s1` que comienza con `s1[0]` y no contiene ninguno de los caracteres encontrados en `s2`.
`strlen`	`size_t strlen (const char *s)` Devuelve la longitud de la cadena `s`.
`strncat()`	`char* strncat(char* s1, const char*s2, size_t n);` Añade los primeros *n* caracteres de `s2` a `s1`. Devuelve `s1`. Si `n >= strlen(s2)`, entonces `strncat(s1, s2, n)` tiene el mismo efecto que `strcat(s1, s2)`.
`strncmp()`	`int strncmp(const char* s1, const char* s2, size_t n);` Compara `s1` con la subcadena `s` de los primeros `n` caracteres de `s2`. Devuelve un entero negativo, cero o un entero positivo, según que `s1` lexicográficamente sea menor, igual o mayor que `s`. Si `n Ø strlen(s2)`, entonces `strncmp(s1, s2, n)` y `strcmp(s1, s2)` tienen el mismo efecto.
`strnset`	`char *strnset(char *s, int ch, size_t n);` Utiliza `strcmp()` sobre una cadena existente para fijar *n* bytes de la cadena al carácter *`ch`*.
`strpbrk()`	`char* strpbrk(const char* s1, cont char* s2);` Devuelve la dirección de la primera ocurrencia de `s1` de cualquiera de los caracteres de `s2`. Devuelve `NULL` si ninguno de los caracteres de `s2` aparece en `s1`.
`strrchr()`	`char* strrchr(const char* s, int c);` Devuelve un puntero a la última ocurrencia de `c` en `s`. Devuelve `NULL` si `c` no está en `s`.
`strspn()`	`size_t strspn(char* s1, const char* s2);` Devuelve la longitud de la subcadena más larga de `s1` que comienza con `s2[0]` y contiene únicamente caracteres encontrados es `s2`.
`strstr`	`char *strstr(const char *s1, const char *s2);` Busca la cadena *`s2`* en *`s1`* y devuelve un puntero a los caracteres donde se encuentra *`s2`*.
`strtok()`	`char* strtok(char* s1, const char* s2);` Analiza la cadena `s1` en *tokens* (componentes léxicos) delimitados por los caracteres encontrados en la cadena `s2`. Después de la llamada inicial `strtok(s1, s2)`, cada llamada sucesiva a `strtok(NULL, s2)` devuelve un puntero al siguiente *token* encontrado en `s1`. Estas llamadas cambian la cadena `s1`, reemplazando cada separador con el carácter `NULL ('\0')`.

Considérese el programa C++ `PASARRAY.CPP`, que implementa una función `Longitud()` que calcula la longitud de una cadena terminada en nulo. El parámetro `cad` se declara como un array de caracteres de tamaño desconocido. Este ejemplo es un caso que muestra el paso de un array por valor, método no recomendado.

```
// PASARRAY.CPP
// Paso de un array como parámetro por valor a una función
// Este método no es eficiente

#include <iostream>
using namespace std;

int Longitud(char cad[])
{
   int cuenta = 0;
   while (cad[cuenta++] != '\0');
   return cuenta;
}

void main(void)
{
   cout << Longitud("C++ es mejor que C") << endl;

   cout << "Pulse Intro(Enter) para continuar";
   cin.get();
}
```

En el programa principal contiene el código para la dirección de la constante de cadena a la función **`Longitud()`**. El cuerpo bucle `while` dentro de la función cuenta los caracteres no nulos y termina cuando se encuentra el byte nulo al final de la cadena.

11.4.1. Uso del operador de referencia para tipos array

Otro método de escribir la función `Longitud()` es utilizar el operador de referencia & de C++. Cuando se utiliza este operador, se pasa por el parámetro por referencia.

```
// PASARREF.CPP
// Paso de un array como parámetro por referencia

#include <iostream>
using namespace std;

typedef char cad80[80];

int Longitud(cad80 &cad)
{
   int cuenta = 0;
   while (cad[cuenta++] != '\0');
      return cuenta;
}

void main(void)
{
```

```
   cad80 s = "C++ es mejor que C";

   cout << Longitud(s) << endl;

   cout << "Pulse Intro(Enter) para continuar";
   cin.get();
}
```

Obsérvese que se ha utilizado `typedef` debido a que el compilador no acepta `char cad[]`.

11.4.2. Uso de punteros para pasar una cadena

Los punteros se pueden utilizar para pasar arrays a funciones.

```
// EXTRAER.CPP
// Uso de punteros cuando se pasan arrays a funciones

#include <iostream>
using namespace std;

// Función extraer copia num_cars caracteres
// de la cadena fuente a la cadena destino
int extraer(char *dest, char *fuente, int num_cars)
{
   int cuenta;
   for(cuenta = 1; cuenta <= num_cars; cuenta++)
      *dest++ = *fuente++;
   *dest = '\0';
   return cuenta;  // devuelve número de caracteres
}

void main(void)
{
   char s1[40] = "Sierra de Cazorla";
   char s2[40];

   extraer(&s2[0], &s1[0], 3);

   cout << s2 << endl;

   cout << "Pulse Intro para continuar";
   cin.get();
}
```

Observe que en las declaraciones de parámetros, ningún array está definido, sino punteros de tipo char. En la línea

```
*dest++ = *fuente++;
```

los punteros se utilizan para acceder a las cadenas fuente y destino, respectivamente. En la llamada a la función `extraer` se utiliza el operador & para obtener la dirección de las cadenas fuente y destino.

11.5. ASIGNACIÓN DE CADENAS

C++ soporta dos métodos para asignar cadenas. Uno de ellos ya se ha visto anteriormente cuando se inicializaban las variables de cadena. La sintaxis utilizada era

```
char VarCadena[LongCadena] = ConstanteCadena;
```

Ejemplo 11.6

```
char Cadena[81] = "C++ versión 3.0";
char nombre[] = "Pepe Mackoy";
```

El segundo método para asignación de una cadena a otra es utilizar la función `strcpy`. La función `strcpy` copia los caracteres de la cadena fuente a la cadena destino. La función supone que la cadena destino tiene espacio suficiente para contener toda la cadena fuente. El prototipo de la función es:

```
char* strcpy(char* destino, const char* fuente);
```

Ejemplo 11.7

```
char nombre[41];
strcpy(nombre, "Cadena a copiar");
```

La función **`strcpy()`** copia `"Cadena a copiar"` en la cadena nombre y añade un carácter nulo al final de la cadena resultante. El siguiente programa muestra una aplicación de `strcpy()`.

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
   char s[100] = "Buenos días Mr. Mackoy", t[100];
   strcpy(t, s);
   strcpy(t+12, "Mr. C++");
   cout << s << endl << t << endl;
}
```

Al ejecutarse el programa produce la salida:

```
Buenos días Mr. Mackoy
Buenos días Mr. C++
```

La expresión `t+12` obtiene la dirección de la cadena s en Buenos días Mr. Mackoy.

11.5.1. La función `strncpy`

El prototipo de la función **`strncpy`** es

```
char* strncpy(char* destino, const char* fuente, size_t num);
```

y su propósito es copiar *num* caracteres de la cadena fuente a la cadena destino. La función realiza truncamiento o rellenado de caracteres si es necesario.

Ejemplo 11.8

```
char cad1[] = "Magina";
char cad2[] = "Hola mundo";
strncpy(cad1, cad2, 6);
```

La variable `cad1` contiene ahora la cadena `"Hola"`.

> ***Consejo***
>
> Los punteros pueden manipular las partes posteriores de una cadena, asignando la dirección del primer carácter a manipular.
>
> ```
> char cad1[41] = "Hola mundo";
> char cad2[41];
> char* p = cad1;
>
> p += 5;     // p apunta a la cadena "mundo"
> strcpy(cad2, p);
> cout << cad2 << "\n";
> ```
>
> La sentencia de salida visualiza la cadena `"mundo"`.

11.6. LONGITUD Y CONCATENACIÓN DE CADENAS

Muchas operaciones de cadena requieren conocer el número de caracteres de una cadena (*longitud*), así como la unión (*concatenación*) de cadenas.

11.6.1. La función `strlen()`

La función **`strlen`** calcula el número de caracteres del parámetro cadena, excluyendo el carácter nulo de terminación de la cadena. El prototipo de la función es

```
size_t strlen(const char* cadena)
```

El tipo de resultado `size_t` representa un tipo entero general.

```
char cad[] = "1234567890";
unsigned i;
i = strlen(cad);
```

Estas sentencias asignan *10* a la variable *i*.

Ejemplo

```
#include <string>
void main()
```

```
{
   char s[] = "ABCDEFG";
   cout << "strlen(" << s << ") = " << strlen(s) << endl;
   cout << "strlen(\ "\") = " << strlen("") << endl;
   char bufer[80];
   cout << "Introduzca cadena:"; cin >> bufer;
   cout << "strlen(" << bufer << ") = " << strlen(bufer)
        << endl;
}
```

Ejecución

```
strlen (ABCDEFGHI) = 9
strlen ("") = 0
Introduzca cadena: Cazorla
strlen(computadora) = 10
```

11.6.2. Las funciones `strcat` y `strncat`

En muchas ocasiones se necesita construir una cadena, añadiendo una cadena a otra cadena, operación que se conoce como *concatenación*. Las funciones **`strcat`** y **`strncat`** realizan operaciones de concatenación.

`strcat` añade el contenido de la cadena fuente a la cadena destino, devolviendo un puntero a la cadena destino. Su prototipo es:

```
char* strcat(char* destino, const char* fuente);
```

Ejemplo 11.9

```
char cadena[81];
strcpy(cadena, "Borland");
strcat(cadena, " C++");
```

La variable `cadena` contiene ahora `"Borland C++"`.

Es posible limitar el número de caracteres a copiar utilizando la función **`strncat`**. La función **`strncat`** añade *`num`* caracteres de la cadena fuente a la cadena destino y devuelve el puntero a la cadena destino. Su prototipo es

```
char* strncat(char* destino, const char* fuente, size_t num)
```

y cuando se invoca con una llamada tal como

```
strncat(t, s, n);
```

`n` representa los primeros `n` caracteres de `s` que se copian, a menos que se encuentre un carácter nulo, en cuyo momento se termina el proceso.

Ejemplo 11.10

```
char cad1[81] = "Hola soy yo ";
char cad2[41] = "Luis Mortimer";
strncat(cad1, cad2, 4);
```

La variable `cad1` contiene ahora `"Hola yo soy Luis"`.

Ejercicio 11.3

El programa añade la cadena `s2` al final de la cadena `s1`.

```
#include <iostream>
#include <string>  // también <cstring>
using namespace std;

void main()
{
   char s1[] = "ABCDEFGH";
   char s2[] = "XYZ";
   cout << "Antes de strcat(s1, s2): \n";
   cout << "\ts1 = [" << s1 << "], longitud = " << strlen(s1) << endl;
   cout << "\ts2 = [" << s2 << "], longitud = " << strlen(s2) << endl;
   strcat(s1, s2);
   cout << "Después de strcat(s1, s2):\n";
   cout << "\ts1 = [" << s1 << "], longitud = " << strlen(s1) << endl;
   cout << "\ts2 = [" << s2 << "], longitud = " << strlen(s2) << endl;
}
```

Ejecución

```
Antes de strcat(s1, s2):
   s1 = [ABCDEFGH], longitud = 8
   s2 = [XYZ], longitud = 3

Después de strcat(s1, s2)
   s1 = [ABCDEFGHXYZ], longitud = 11
   s2 = [XYZ], longitud = 3
```

11.7. COMPARACIÓN DE CADENAS

Dado que las cadenas son arrays de caracteres, la biblioteca `string.h` proporciona un conjunto de funciones que comparan cadenas. Estas funciones comparan los caracteres de dos cadenas utilizando el valor ASCII de cada carácter. Las funciones de comparación son: **`strcmp`**, **`stricmp`**, **`strncmp`** y **`strnicmp`**.

11.7.1. La función `strcmp`

Cuando se desea determinar si una cadena es igual a otra, o mayor o menor que otra, se debe utilizar la función **`strcmp()`**.

`strcmp()` compara su primer parámetro con su segundo, y devuelve 0 si las dos cadenas son idénticas, un valor menor que cero si la cadena 1 es menor que la cadena 2, o un valor mayor que cero si la cadena 1 es mayor que la cadena 2 (los términos *«mayor que»* y *«menor que»* se refieren a la ordenación alfabética de las cadenas). Por ejemplo, Alicante es menor que Sevilla. Así, la letra *A* es menor que la letra *a*, la letra *Z* es menor que la letra *a*. El prototipo de la función **`strcmp`** es

```
int strcmp(const char* cad1, const char* cad2);
```

La función compara las cadenas *cad1* y *cad2*. El resultado entero es:

< 0	*si*	`cad1`	*es menor que*	`cad2`
= 0	*si*	`cad1`	*es igual a*	`cad2`
> 0	*si*	`cad1`	*es mayor que*	`cad2`

Ejemplo 11.11

```
char cad1[] = "Microsoft C++";
char cad2[] = "Microsoft Visual C++"
int i;

i = strcmp(cadena1, cadena2);  // i, toma un valor negativo

strcmp("Waterloo", "Windows")  < 0
strcmp("Mortimer", "Mortim")> 0
strcmp("Jeronimo", "Jeronimo") = 0
```

La comparación se realiza examinando los primeros caracteres de `cad1` y `cad2`; a continuación, los siguientes caracteres, y así sucesivamente. Este proceso termina cuando:

- se encuentran dos caracteres distintos del mismo orden: `cad1[i]` y `cad2[i]`;
- se encuentra el carácter nulo en `cad[i]` o `cad2[i]`

`Waterloo`	*es menor que*	`Windows`
`Mortimer`	*es mayor que*	`Mortim_caracter nulo`
`Jeronimo`	*es igual que*	`Jeronimo`

11.7.2. La función `stricmp`

La función **`stricmp`** compara las cadenas `cad1` y `cad2` sin hacer distinción entre mayúsculas y minúsculas. El prototipo es

```
int stricmp(const char* cad1, const char* cad2);
```

Ejemplo 11.12

```
char cad1[] = "Turbo C++";
char cad2[] = "TURBO C++";
int i;

i = stricmp(cad1, cad2); // se asigna 0 a la variable i ya que las
                         // dos cadenas son iguales
```

11.7.3. La función `strncmp`

La función **`strncmp`** compara los *`num`* caracteres más a la izquierda de las dos cadenas `cad1` y `cad2`. El prototipo es

```
int strncmp(const char* cad1, const char* cad2, size_t num);
```

y el resultado de la comparación será:

`< 0`	*si*	`cad1`	*es menor que*	`cad2`
`= 0`	*si*	`cad1`	*es igual que*	`cad2`
`> 0`	*si*	`cad1`	*es mayor que*	`cad2`

Ejemplo 11.13

```
char cadena1[] = "Turbo C++";
char cadena2[] = "Turbo Prolog"
int i;

i = strncmp(cadena1, cadena2, 7);
```

Esta sentencia asigna un número negativo a la variable *i*, ya que `"Turbo C"` es menor que `"Turbo P"`.

11.7.4. La función `strnicmp`

La función **`strnicmp`** compara los caracteres `num` a la izquierda en las dos cadenas, `cad1` y `cad2`, con independencia del tamaño de las letras. El prototipo es

```
int strnicmp(const char* cad1, const char* cad2, size_t num);
```

El resultado será:

`< 0`	*si*	`cad1`	*es menor que*	`cad2`
`= 0`	*si*	`cad1`	*es igual que*	`cad2`
`> 0`	*si*	`cad1`	*es mayor que*	`cad2`

Ejemplo 11.14

```
char cadena1[] = "Turbo C++";
char cadena2[] = "TURBO C++";
int i;

i = strnicmp(cadena1, cadena2, 5);
```

Esta sentencia asigna `0` a la variable `i`, ya que las cadenas `"Turbo"` y `"TURBO"` difieren sólo en el tamaño de sus caracteres.

11.8. INVERSIÓN DE CADENAS

La biblioteca `string.h` incluye la función **`strrev`** que sirve para invertir los caracteres de una cadena. Su prototipo es:

```
char *strrev(char *s);
```

`strrev` invierte el orden de los caracteres de la cadena especificada en el argumento *s*; devuelve un puntero a la cadena resultante.

Ejemplo 11.15

```
char cadena[] = "Hola";

strrev(cadena);
cout << cadena;     // visualiza "aloH"
```

El programa `INVERTIR.CPP` invierte el orden de la cadena `Hola mundo`.

```
#include <iostream>
#include <string>

int main(void)
{
   char *cadena = "Hola mundo";
   char *resultado;
   strrev(cadena);
   cout << "Cadena inversa: %s\n", cadena);

   return 0;
}
```

11.9. CONVERSIÓN DE CADENAS

La biblioteca `string.h` de la mayoría de los compiladores C++ suele incluir funciones para convertir los caracteres de una cadena a letras mayúsculas y minúsculas respectivamente. Estas funciones se llaman **`strlwr`** y **`strupr`** en compiladores AT&T y Borland, mientras que en compiladores de Microsoft se denominan **`_strlwr`** y **`_strupr`**.

11.9.1. Función `strupr`

La función **`strupr`** convierte las letras minúsculas de una cadena a mayúsculas. Su prototipo es:

```
char *strupr(char *s);
```

Ejemplo 11.16

```
// archivo MAYUSUNO.CPP

#include <iostream>
usign namespace std;
```

```
#include <string>

int main(void)
{
   char *cadena = "abcdefg";

   strupr(cadena);
   cout << "La cadena convertida es:" << cadena;

   return 0;
}
```

11.9.2. Función `strlwr`

La función **`strlwr`** convierte las letras mayúsculas de una cadena a letras minúsculas.

```
// archivo MINUS.CPP

#include <iostream>
using namespace std;

#include <string>

int main(void)
{
   char *cadena = "ABCDEFG";

   strlwr(cadena);
   cout << "La cadena convertida es:" << cadena;

   return 0;
}
```

11.10. CONVERSIÓN DE CADENAS A NÚMEROS

Es muy frecuente tener que convertir números almacenados en cadenas de caracteres a tipos de datos numéricos. C++ proporciona las funciones **`atoi`**, **`atof`** y **`atol`**, que realizan estas conversiones. Estas tres funciones se incluyen en la biblioteca `<cstdlib>` por lo que ha de incluir en su programa la directiva

```
#include <cstdlib>.
```

11.10.1. Función `atoi`

La función **`atoi`** convierte una cadena a un valor entero. Su prototipo es:

```
int atoi(const char *cad);
```

`atoi` convierte la cadena apuntada por `cad` a un valor entero. La cadena debe de tener la representación de un valor entero y el formato siguiente:

[espacio en blanco] [signo] [ddd]

[espacio en blanco] = cadena opcional de tabulaciones y espacios

```
[signo]          =  un signo opcional para el valor
[ddd]            =  la cadena de dígitos
```

Una cadena que se puede convertir a un entero es:

```
"1232"
```

Sin embargo, la cadena siguiente no se puede convertir a un valor entero:

```
"-1234596.495"
```

La cadena anterior se puede convertir a un número de coma flotante con la función **`atof()`**.

> Si la cadena no se puede convertir, **`atoi()`** devuelve cero.

Ejemplo 11.17

```
char *cadena = "453423";

valor = atoi(cadena);
```

11.10.2. Función `atof`

La función **`atof`** convierte una cadena a un valor de coma flotante. Su prototipo es:

```
double atof(const char *cad);
```

`atof` convierte la cadena apuntada por `cad` a un valor `double` en coma flotante. La cadena de caracteres debe tener una representación de caracteres de un número de coma flotante. La conversión termina cuando se encuentre un carácter no reconocido. Su formato es:

```
[espacio en blanco][signo][ddd][.][ddd][eE][signo][ddd]
```

Ejemplo 11.18

```
char *cadena = "545.7345";

valor = atof(cadena);
```

11.10.3. Función `atol`

La función **`atol`** convierte una cadena a un valor largo (`long`). Su prototipo es

```
long atol(const char *cad);
```

La cadena a convertir debe tener un formato de valor entero largo:

```
[espacio en blanco] [signo] [ddd]
```

EJEMPLO 11.19

```
char *cadena = "45743212";

valor = atol(cadena);
```

11.11. BÚSQUEDA DE CARACTERES Y CADENAS

La biblioteca `string.h` contiene un número de funciones que permiten localizar caracteres en cadenas y subcadenas en cadenas.

Funciones de búsqueda de caracteres	`strchr` `strcspn`	`strrchr` `strpbrk`	`strspn`
Funciones de búsqueda de cadenas	`strstr`	`strtok`	

11.11.1. La función `strchr()`

El prototipo de la función `strchr` es

```
char *strchr(const char *s, int c);
```

`strchr` permite buscar caracteres y patrones de caracteres en cadenas; localiza la primera ocurrencia de un carácter *c* en una cadena s. La búsqueda termina en la primera ocurrencia de un carácter coincidente.

Ejemplo 11.20

```
char cad[81] = "Turbo C++";
char *cadPtr;

cadPtr = strchr(cad, '+');
```

11.11.2. La función `strrchr()`

La función **`strrchr`** localiza la última ocurrencia del patrón *c* en la cadena s. La búsqueda termina con la primera ocurrencia de un carácter coincidente. Si no se encuentra el carácter *c* en la cadena s, la función produce un resultado `NULL`. Su prototipo es

```
char *strchr(const char *s, int c);
```

carácter buscado

cadena de búsqueda

Un ejemplo de aplicación de esta función es:

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main(void)
{
   char *cadena = "----x----";
   char *resultado;

   resultado = strchr(cadena, 'x');
   cout << "Valor devuelto:" << resultado;
}
```

11.11.3. La función `strspn()`

La función **`strspn`** devuelve el número de caracteres de la parte izquierda de una cadena *destino* `s1` que coincide con cualquier carácter de la cadena *patrón* `s2`. El prototipo de **`strspn`** es

```
size_t strspn(const char *s1, const char *s2);
```

El siguiente ejemplo busca el segmento de `cadena1` que tiene un subconjunto de `cadena2`.

```
#include <iostream>
#include <string>

int main(void)
{
   char *cadena1 = "123456";
   char *cadena2 = "abc123";
   int longitud;
   longitud = strspn(cadena1, cadena2);
   cout << "Longitud =" << longitud;
   return 0;
}
```

Al ejecutarse, escribirá en pantalla

```
Longitud = 3
```

11.11.4. La función `strcspn`

La función **`strcspn`** encuentra una subcadena dentro de una cadena. Devuelve el índice del primer carácter de la primera cadena que está en el conjunto de caracteres especificado en la segunda cadena. El prototipo de **`strcspn`** es:

```
size_t strcspn(const char *s1, const char *s2);
```

Ejemplo 11.21

```
char cadena[] = "Los manolos de Carchelejo";
int i;

i = strcspn(cadena, " de");
```

El ejemplo anterior asigna `11` (la longitud de `"Los manolos"`) a la variable `i`.

11.11.5. La función `strpbrk`

La función **`strpbrk`** recorre una cadena buscando caracteres pertenecientes a un conjunto de caracteres especificado. El prototipo es

```
char *strpbrk(const char *s1, const char *s2);
```

Esta función devuelve un puntero a la primera ocurrencia de cualquier carácter de `s2` en `s1`. Si las dos cadenas no tienen caracteres comunes se devuelve `NULL`.

```
char *cad = "Hello Dolly, hey Julio";
char *subcad = "hy";
char *ptr;

ptr = strpbrk(cad, subcad);
cout << ptr << "\n";
```

El segmento de programa visualiza `"hey Julio",` ya que `"h"` se encuentra en la cadena antes que la `"y"`.

11.11.6. La función `strstr`

La biblioteca `string.h` contiene las funciones **`strstr`** y **`strtok`**, que permiten localizar una subcadena en una cadena o bien romper una cadena en subcadenas.

La función **`strstr`** busca una cadena dentro de otra cadena. El prototipo de la función es

```
char *strstr(const char *s1, const char *s2);
```

La función devuelve un puntero al primer carácter de la cadena `s1` que coincide con la cadena `s2`. Si la subcadena `s1` no está en la cadena `s2,` la función devuelve `NULL`.

```
char *cad1 = "123456789";
char *cad2 = "456";
char *resultado;

resultado = strstr(cad1, cad2);
cout << resultado << "\n";
```

El segmento de programa anterior visualiza `456789`.

11.11.7. La función `strtok`

La función **`strtok`** permite romper una cadena en subcadenas, basada en un conjunto especificado de caracteres de separación. Su prototipo es

```
char *strtok(char *s1, const char *s2);
```

`strtok` lee la cadena `s1` como una serie de cero o más símbolos y la cadena `s2` como el conjunto de caracteres que se utilizan como separadores de los símbolos de la cadena `s1`. Los símbolos en la ca-

dena `s1` pueden encontrarse separados por un carácter o más del conjunto de caracteres separadores de la cadena `s2`.

```
#include <iostream>
using namespace std;

#include <string>

void main()
{
   char *cad = "(Pepe_Luis + Mortimer) * Mackoy";
   char *separador = "+ * ()";
   char *ptr = cad;
   cout << cad;
   ptr = strtok(cad, separador);
   cout << "se rompe en" << ptr;
   while (ptr) {
      cout << ptr;
      ptr = strtok(NULL, separador);
   }
}
```

Este ejemplo visualiza lo siguiente cuando se ejecuta el programa:

```
(Pepe_Luis * Mortimer) * Mackoy
```

La cadena anterior se rompe en tres subcadenas:

```
Pepe_Luis, Mortimer, Mackoy
```

RESUMEN

En este capítulo se han examinado las funciones de manipulación de cadenas incluidas en el archivo de cabecera `string.h`. Los temas tratados han sido:

- Las cadenas en C++ son arrays de caracteres que terminan con el carácter nulo (el carácter 0 de ASCII).
- La entrada de cadenas requiere el uso de la función **`getline`**.
- La biblioteca `string.h` contiene numerosas funciones de manipulación de cadenas; entre ellas, se destacan las funciones que soportan asignación, concatenación, conversión, inversión y búsqueda.
- C++ soporta dos métodos de asignación de cadenas. El primer método asigna una cadena a otra, cuando se declara esta última. El segundo método utiliza la función **`strcpy`**, que puede asignar una cadena a otra en cualquier etapa del programa.
- La función **`strlen`** devuelve la longitud de una cadena.
- Las funciones **`strcat`** y **`strncat`** permiten concatenar dos cadenas. La función **`strncat`** permite especificar el número de caracteres a concatenar.
- Las funciones **`strcmp, stricmp, strncmp`** y **`strnicmp`** permiten realizar diversos tipos de comparaciones. Las funciones **`strcmp`** y **`stricmp`** realizan una comparación de dos cadenas, sin tener en cuenta el tamaño de las letras. La función `strncmp` es una variante de la función `strcmp`, que utiliza un número especificado de caracteres al comparar las cadenas. La función **`strnicmp`** es una versión de la función `strncmp` que realiza una versión con independencia del tamaño de las letras.
- Las funciones **`strlwr`** y **`strupr`** convierte los caracteres de una cadena en letras minúsculas y mayúsculas respectivamente.
- La función **`strrev`** invierte el orden de caracteres en una cadena.
- Las funciones **`strchr, strspn, strcspn`** y **`strpbrk`** permiten buscar caracteres y patrones de caracteres en cadenas.
- La función **`strstr`** busca una cadena en otra cadena. La función **`strtok`** *rompe* (divide) una cadena en cadenas más pequeñas (subcadenas) que se separan por caracteres separadores especificados.

Asimismo, se han descrito las funciones de conversión de cadenas de tipo numérico a datos de tipo numérico. C++ proporciona las siguientes funciones de conversión: **`atoi(s), atol(s)`** y **`atof(s),`** que convierten el argumento *s* (cadena) a enteros, enteros largos y reales de coma flotante.

EJERCICIOS

11.1. Suponga que no existe la función `strlen,` el código de una función que realizará la tarea de dicha función.

11.2. Escribir un programa que lea palabras de un búfer de 79 caracteres.

11.3. ¿Qué está mal cuando se usa la sentencia

```
cin <<< s ;
```

para leer la entrada `"!Hola, Mundo¡"` en una cadena `s`?

11.4. ¿Cuál es la diferencia entre las dos siguientes sentencias, si `s1` y `s2` tienen tipos `*char`?

```
s1 = s2 ;
strcpy(s1, s2) ;
```

11.5. ¿Qué hace el siguiente código?

```
char *s1 = "ABCDE" ;
char *s2 = "ABC" ;
if(strcmp(s1, s2)<0) cout << s1 <<
   " < " << s2 << endl ;
else cout << s1 << " >= " s2 <<
   endl ;
```

PROBLEMAS

11.1. Escribir un programa que haga eco de la entrada, línea a línea.

11.2. Contar el número de ocurrencias de la letra `'e'` en el flujo de entrada.

11.3. Escribir un programa que haga eco del flujo de entrada y luego ponga en mayúscula la primera letra de cada palabra.

11.4. Escribir una función que extraiga los enteros del flujo de entrada.

11.5. Leer una secuencia de cadenas, almacenarlas en un array y, a continuación, visualizarla.

11.6. Leer una secuencia de nombres, uno por línea, terminado por el valor centinela ´$´. A continuación, visualizar los nombres almacenados en el array `nombres`.

11.7. Escribir la función `strncat()`.

11.8. Escribir una función que devuelva la forma plural de la palabra española que se le pase.

11.9. Escribir un programa que lea una secuencia de nombres, uno por línea, los ordene y, a continuación, los presente en pantalla.

11.10. Escribir un programa que haga uso de una función `invertir()` para invertir una cadena leída del teclado.

11.11. Leer una línea de entrada. Descartar todos los símbolos excepto los dígitos. Convertir la cadena de dígitos en un entero y fijar el valor del entero a la variable `n`.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág, 195).

2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

11.1. ¿Cuál de las siguientes declaraciones son equivalentes?

```
char var_cad0[10] ="Hola";
char var_cad1[10] = {
  'H','o','l','a'};
char var_cad2[10]= {
  'H','o','l','a','\0'};
char var_cad3[5]= "Hola";
char var_cad4[]= "Hola";
```

11.2. ¿Qué hay de incorrecto (si existe) en el siguiente código?

```
char var_cad[]= "Hola";
      strcat(var_cad, " y adios");
      cout << var_cad<< endl;
```

11.3. Suponga que no existe el código de la función `strlen`. Escriba el código de una función que realice la tarea indicada.

11.4. ¿Qué diferencias y analogías existen entre las variables c1, c2, c3? La declaración es:

```
char** c1;
char* c2[10];
char* c3[10][21];
```

11.5. Escribir un programa que lea dos cadenas de caracteres, las visualice junto con su longitud, las concatene y visualice la concatenación y su longitud.

11.6. La carrera de sociología tiene un total de N asignaturas. Escribir una función que lea del dispositivo estándar de entrada las N asignaturas con sus correspondientes códigos.

11.7. Añadir al Ejercicio 6 funciones para visualizar las N asignaturas y modificar una asignatura determinada.

11.8. Escribir un programa que lea una cadena de caracteres de la entrada y la invierta.

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 198).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

11.1. La función `atoi( )` transforma una cadena formada por dígitos decimales en el equivalente número entero. Escribir una función que transforme una cadena formada por dígitos hexadecimales en un entero largo.

11.2. Definir un array de cadenas de caracteres para poder leer un texto compuesto por un máximo de 80 caracteres por líneas. Escribir una función para leer el texto, y otra para escribirlo; las funciones deben tener dos argumentos, uno el texto y el segundo el número de líneas.

11.3. Se sabe que en las 100 líneas que forman un texto hay valores numéricos enteros, que representan los kg de patatas recogidos en una finca. Los valores numéricos están separados de las palabras por un blanco, o el carácter fin de línea. Escribir un programa que lea el texto y obtenga la suma de todos los valores numéricos.

11.4. Escribir una función que tenga como entrada una cadena y devuelva el número de vocales, de consonantes y de dígitos de la cadena.

11.5. Escribir un programa que encuentre dos cadenas introducidas por teclado que sean anagramas. Se considera que dos cadenas son anagramas si contienen exactamente los mismos caracteres en el mismo o en diferente orden. Hay que ignorar los blancos y considerar que las mayúsculas y las minúsculas son iguales.

11.6. Escribir un programa para las líneas de un texto sabiendo que el máximo de caracteres por línea es 80 caracteres. Contar el número de palabras que tiene cada línea, así como el número total de palabras leídas.

11.7. Se tiene un texto formado por un máximo de 30 líneas, del cual se quiere saber el número de apariciones de una palabra clave. Escribir un programa que lea la palabra clave determine el número de apariciones en el texto.

11.8. Se tiene un texto de 40 líneas. Las líneas tienen un número de caracteres variable. Escribir un programa para almacenar el texto en una matriz de líneas, ajustada la longitud de cada línea al número de caracteres. El programa debe leer el texto, almacenarlo en la estructura matricial y escribir por pantalla las líneas en orden creciente de su longitud.

11.9. Escribir un programa que lea una cadena clave y un texto de, como máximo, 50 líneas. El programa debe eliminar las líneas que contengan la clave.

11.10. Escribir una función que reciba como parámetro una cadena de caracteres y la invierta, sin usar la función `strrev`.

11.11. Escribir una función que reciba una cadena de caracteres, una longitud lon, y un carácter `ch`. La función debe retornar otra cadena de longitud `lon`, que contenga la cadena de caracteres y si es necesario, el carácter `ch` repetido al final de la cadena las veces que sea necesario.

11.12. Se quiere sumar números grandes que no puedan almacenarse en variables de tipo `long`. Por esta razón, se ha pensado en introducir cada número como una cadena de caracteres y realizar la suma extrayendo los dígitos de ambas cadenas. Hay que tener en cuenta que la cadena suma puede tener un carácter más que la máxima longitud de los sumandos.

11.13. Escribir una función que reciba como parámetros un número grande como cadena de caracteres, y lo multiplique por un dígito, que reciba como parámetro de tipo carácter.

11.14. Escribir una función que multiplique dos números grandes, recibidos como cadenas de caracteres.

11.15. Un texto está formado por líneas de longitud variable. La máxima longitud es de 80 caracteres. Se quiere que todas las líneas tengan la misma longitud, la de la cadena más larga. Para ello se debe rellenar con blancos por la derecha las líneas hasta completar la longitud requerida. Escribir un programa para leer un texto de líneas de longitud variable y formatear el texto para que todas las líneas tengan la longitud de la máxima línea.

CAPÍTULO 12

Ordenación y búsqueda

Contenido

INTRODUCCIÓN

Muchas actividades humanas requieren que a diferentes colecciones de elementos utilizados se pongan en un orden específico. Las oficinas de correo y las empresas de mensajería ordenan el correo y los paquetes por códigos postales con el objeto de conseguir una entrega eficiente; los anuarios o listines telefónicos se ordenan por orden alfabético de apellidos con el fin último de encontrar fácilmente el número de teléfono deseado. Los estudiantes de una clase de la universidad se ordenan por sus apellidos o por los números de expediente. Por esta circunstancia una de las tareas que realizan más frecuentemente las computadoras en el procesamiento de datos es la *ordenación.*

El estudio de diferentes métodos de ordenación es una tarea intrínsecamente interesante desde un punto de vista teórico y, naturalmente, práctico. Este capítulo estudia los algoritmos y técnicas de ordenación más usuales y su implementación en C++. De igual modo se estudiará el análisis de los diferentes métodos de ordenación con el objeto de conseguir la máxima eficiencia en su uso real.

En este capítulo se analizarán los métodos básicos y los más avanzados empleados en programas profesionales.

CONCEPTOS CLAVE

- Algoritmos de ordenación básicos.
- Análisis de los algoritmos de búsqueda.
- Búsqueda en listas: búsqueda secuencial y búsqueda binaria.
- Ordenación por burbuja.
- Ordenación por inserción.
- Ordenación por intercambio.
- Ordenación por selección.

12.1. ALGORITMOS DE ORDENACIÓN BÁSICOS

Existen diferentes algoritmos de ordenación elementales o básicos, cuyos detalles de implementación se pueden encontrar en diferentes libros de algoritmos. La enciclopedia de referencia es [Knuth, 1973][1] y sobre todo la segunda edición publicada en el año 1998 [Knuth, 1998][2]. Los algoritmos presentan diferencias entre ellos que los convierten en más o menos eficientes y prácticos según sea la rapidez y eficiencia demostrada por cada uno de ellos. Los algoritmos básicos de ordenación más simples y clásicos son:

- Ordenación por selección.
- Ordenación por inserción.
- Ordenación por burbuja.

Los métodos más recomendados son el de *selección* y el de *inserción*, aunque estudiaremos el método de *burbuja*, por aquello de ser el más sencillo aunque a la par también es el más *ineficiente*; por esta causa no recomendamos su uso, pero sí conocer su técnica.

Los datos se pueden almacenar en memoria central o en archivos de datos externos guardados en unidades de almacenamiento magnético (discos, cintas, disquetes, CD-ROM, DVD, etc.). Cuando los datos se guardan en listas y en pequeñas cantidades, se suelen almacenar de modo temporal en arrays y registros; estos datos se almacenan exclusivamente para tratamientos internos que se utilizan para gestión masiva de datos y se guardan en arrays de una o varias dimensiones. Los datos, sin embargo, se almacenan de modo permanente en archivos y bases de datos que se guardan en discos y cintas magnéticas.

Así pues, existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. Los métodos de ordenación se conocen como *internos* o *externos,* según que los elementos a ordenar estén en la memoria principal o en la memoria externa.

Las técnicas que se analizarán a continuación considerarán, esencialmente, la ordenación de elementos de una lista (*array*) en orden ascendente. En cada caso se desarrollará la eficiencia computacional del algoritmo.

Con el objeto de facilitar el aprendizaje del lector y aunque no sea un método utilizado por su poca eficiencia se describirá en primer lugar el método de ordenación por intercambio con un programa completo que manipula la correspondiente función `OrdIntercambio`, por la sencillez de su técnica y con el objetivo de que el lector no introducido en los métodos de ordenación pueda comprender su funcionamiento y luego pueda asimilar más eficazmente los tres métodos básicos ya citados y los avanzados que se estudiarán más adelante.

12.2. ORDENACIÓN POR INTERCAMBIO

El algoritmo de ordenación tal vez más sencillo sea el denominado de *intercambio,* que ordena los elementos de una lista en orden ascendente. El algoritmo se basa en la lectura sucesiva de la lista a ordenar, comparando el elemento inferior de la lista con los restantes y efectuando intercambio de posiciones cuando el orden resultante de la comparación no sea el correcto.

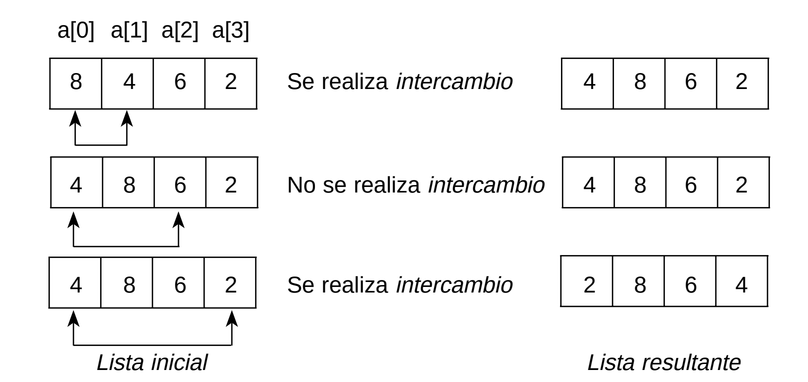
El algoritmo se ilustra con la lista original 8, 4, 6, 2 que ha de convertirse en la lista ordenada 2, 4, 6, 8. El algoritmo realiza *n-1* pasadas (3 en el ejemplo), realizando las siguientes operaciones:

---

[1] [Knuth, 1973] Donald E. Knuth, *The Art of Computer Programming.* vol. 3: *Sorting and Searching*, Addison-Wesley, 1973.

[2] [Knuth, 1998] Donald E. Knuth, *The Art of Computer Programming.* vol 3: *Sorting and Searching*, 2.ª ed., Addison-Wesley, 1998.

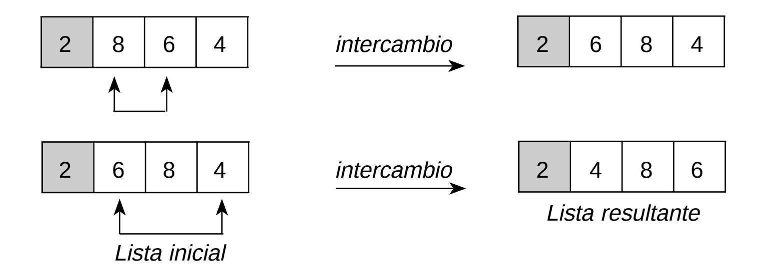
Pasada 0



El elemento de índice 0 (`a[0]`) se compara con cada elemento posterior de la lista de índices 1, 2 y 3. En cada comparación se comprueba si el elemento siguiente es más pequeño que el elemento de índice 0, en ese caso se intercambian. Después de terminar todas las comparaciones el elemento más pequeño se localiza en el índice 0.

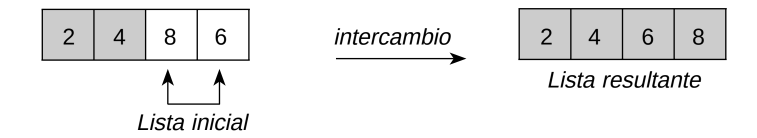
Pasada 1

El elemento más pequeño ya está localizado en el índice 0, y se considera la sublista restante 8, 6, 4. El algoritmo continúa comparando el elemento de índice 1 con los elementos posteriores de índices 2 y 3. Por cada comparación, si el elemento mayor está en el índice 1 se intercambian los elementos. Después de hacer todas las comparaciones, el segundo elemento más pequeño de la lista se almacena en el índice 1.



Pasada 2

La sublista a considerar ahora es 8, 6 ya que 2, 4 está ordenada. Una comparación única se produce entre los dos elementos de la sublista.



La función `OrdIntercambio` utiliza bucles anidados. Suponiendo que la lista es de tamaño *n*, el rango del bucle externo irá desde el índice 0 hasta *n*-2. Por cada índice *i*, se comparan los elementos posteriores de índices *j = i +1, i +2, ..., n*-1. La comparación y el intercambio (*swap*) de los elementos

`a[i], a[j]` se realiza con la función `Intercambio` que utiliza el algoritmo siguiente para intercambiar entre sí los valores `x` e `y`.

```
aux = x;
x = y;
y = aux ;
```

Ejemplo 12.1

El programa `OrdSwap.cpp` ordena una lista de 20 elementos desordenados introducida en un array y posteriormente la imprime (o visualiza) en pantalla

```
#include <iostream>
using namespace std;

// intercambia los valores de dos variables enteras x e y.
void Intercambio (int &x, int &y)
{
   int aux = x;      // almacena valor original de x

   x = y;            // reemplaza x por y
   y = aux;          // asigna a y el valor original de x
}

// ordena los elementos del array de enteros en orden
// ascendente
void OrdIntercambio (int a[], int n)
{
   int i, j ;

   // se realizan n-1 pasadas
   // a[0], ... , a[n-2]
   for (i = 0 ; i < n-1 ; i++)
      // se coloca el mínimo de a[i+1]...a[n-1] en a[i]
      for (j = i+1 ; j < n ; j++)
         // intercambio si a[i] > a[j]
         if (a[i] > a[j])
            Intercambio (a[i], a[j]) ;
}

// Imprimir la lista en orden ascendente
void ImprimirLista (int a[], int n)
{
   for (int i = 0 ; i < n ; i++)
      cout << a[i] << "  ";
   cout << endl ;
}

void main(void)
{
   int lista[20] = { 30, 35, 38, 58, 14, 15, 50, 27, 10, 20,
                     12, 85, 49, 65, 86, 60, 25, 90, 5, 16 }
   int i ;

   cout << "Lista original \n";
   ImprimirLista(lista, 20);
```

```
   OrdIntercambio(lista, 20);
   cout << endl << "Lista ordenada "<< endl;
   ImprimirLista (lista, 20);
}
```

La ejecución del programa `OrSwap` produce

```
Lista original
30 35 38 58 14 15 50 27 10 20 12 85 49 65 85 60 25 90 5 16

Lista ordenada
5 10 12 14 15 16 20 25 27 30 35 38 49 50 58 60 65 85 86 90
```

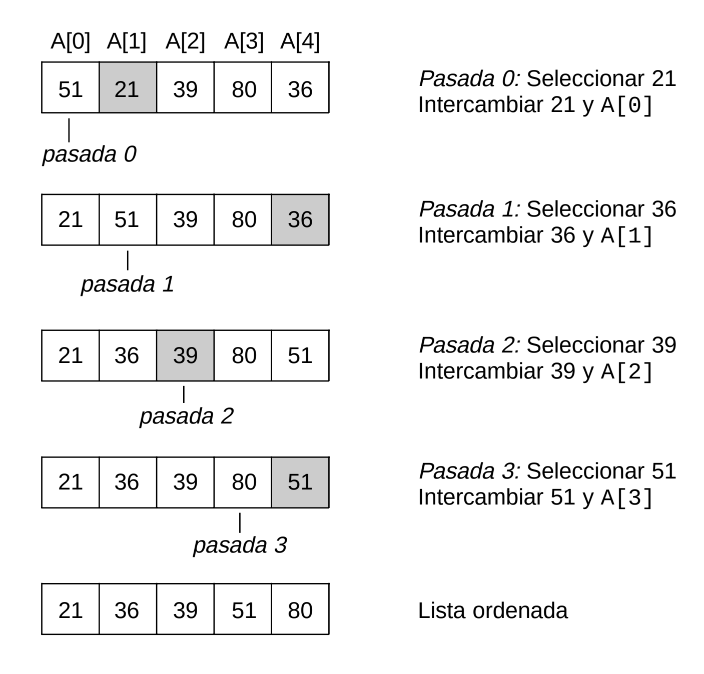
12.3. ORDENACIÓN POR SELECCIÓN

Considérese el algoritmo para ordenar un array `A` de enteros en orden ascendente, es decir del número más pequeño a la posición menor. Si el array `A` tiene *n* elementos, se trata de reordenar los valores del array de modo que el dato contenido en `A[0]` sea el valor más pequeño, el valor almacenado en `A[1]` el siguiente más pequeño, y así hasta `A[n-1]` que ha de contener el elemento de mayor valor. El algoritmo se apoya en sucesivas pasadas que intercambian el elemento más pequeño sucesivamente con el primer elemento de la lista `A[0]`. Así, se busca el elemento más pequeño de la lista y se intercambia con `A[0]`, primer elemento de la lista.

```
[A]      A[0]  A[1]  A[2].... A[n-1]
```

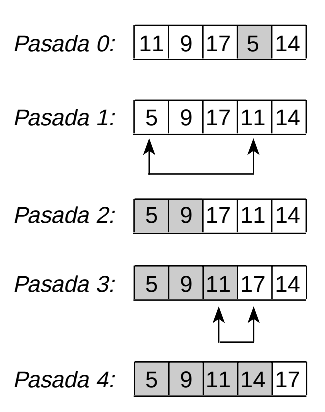
Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista `A[1], A[2]...A[n-1]` permanece desordenada. La siguiente pasada busca en esta lista desordenada y *selecciona* el elemento más pequeño, que se almacena entonces en la posición `A[1]`. De este modo los elementos `A[0]` y `A[1]` están ordenados y la sublista `A[2], A[3]...A[n-1]`; entonces, se selecciona el elemento más pequeño y se intercambia con `A[2]`. El proceso continúa n-1 pasadas en cuyo momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el array completo ha quedado ordenado.

Un ejemplo práctico ayudará a la comprensión del algoritmo. Consideremos un array `A` con 5 valores enteros `51, 21, 39, 80, 36`:



Ejemplo 12.2

Ordenar por selección la lista de enteros 11, 9, 17, 5, 14.



12.3.1. Algoritmo de selección

Los pasos del algoritmo son:

- Seleccionar el elemento más pequeño de la lista `A`. Intercambiarlo con el primer elemento `A[0]`. Ahora la entrada más pequeña está en la primera posición del vector.
- Considerar las posiciones de la lista `A[1], A[2], A[3]`..., seleccionar el elemento más pequeño e intercambiarlo con `A[1]`. Ahora las dos primeras entradas de `A` están en orden.
- Continuar este proceso encontrando o seleccionando el elemento más pequeño de los restantes elementos de la lista, intercambiándolos adecuadamente.

El método de ordenación se basa en seleccionar la posición del elemento más pequeño del array y su colocación en la posición que le corresponde. El algoritmo de selección se apoya en sucesivas pasadas que intercambian el elemento más pequeño sucesivamente con el primer elemento de la lista. El algoritmo de ordenación por selección de una lista (vector) de `n` elementos se realiza de la siguiente forma: se encuentra el elemento menor de la lista y se intercambia el elemento menor con el elemento de subíndice `1`. Después, se busca el elemento menor en la sublista `2 .. n – 1` y se intercambia con la posición `2`. El proceso continúa sucesivamente, durante `n–1` pasadas. Una vez terminadas las pasadas la lista desordenada se reduce a un elemento (el mayor de la lista) y el array completo ha quedado ordenado.

Ejemplo 12.3

Codificación del método de ordenación por selección.

```
void seleccion(float A[], int n)
{
   int i, j, indicemin;
   float auxiliar;

for (i = 0; < n – 1; i++)
{
   indicemin = i;                              // posicion del menor
   for(j = i + 1; j < n; j++
```

```
        if(A[j] < A[indicemin])
           indicemin = j;                      // nueva posicion del menor

        auxiliar = A[indicemin];               // intercambia posiciones
        A[indicemin] = A[i];
        A[i] = auxiliar;
     }
  }
```

El proceso de selección `OrdSeleccion` ordena una lista o vector de enteros. En la pasada *I*, el proceso de selección explora la sublista `A[I]` a `A[n-1]` y fija el índice menor como el elemento más pequeño. Después de terminar la exploración, los elementos `A[I]` y `A[indiceMenor]` intercambian las posiciones. La función `OrdSeleccion` y la utilidad `Intercambio` se almacenan en el archivo `"ordlista.h"` haciendo uso de una función de plantilla.

```
// ordenar un array de n elementos de tipo T
// utilizando el algoritmo de ordenación por selección

template <class T>
void OrdSeleccion (T A[], int n)
{
   // índice del elemento menor en cada pasada
   int IndiceMenor;
   int i, j;

   // ordenar A[0]..A[n-2] y A[n-1] en cada pasada
   for (i = 0; i < n-1; i++)
   {
     // comienzo de la exploración en índice i
     // fija IndiceMenor a i
      IndiceMenor = i;
     // j explora la sublista A[i+1]..A[n-1]
     for (j = i + 1; j < n; j++)
      // actualiza IndiceMenor si se encuentra elemento más pequeño
         if (A[j] < A[IndiceMenor]
            IndiceMenor = j;
      // Cuando se termina, situar el elemento más pequeño en A[i]

      Intercambio (A[i], A[IndiceMenor]);
   }
}
```

Recordemos que la declaración de la función de plantilla (también llamadas plantillas de funciones) comienza con una lista de parámetros de la plantilla con el formato

```
template < class T1, class T2,...>
```

En nuestro caso, sólo se tiene un tipo de dato genérico `T` que se pasa como parámetro cuando la función plantilla se utiliza. Es decir, la declaración de la función es

```
template <classT >  // T es un tipo de dato genérico
void OrdSeleccion (T A[], int n)
```

```
{
   ...
}
```

El algoritmo `OrdSeleccion` requiere una función `Intercambio` que realice el intercambio de dos elementos *x* e *y*:

```
//intercambiar los valores de dos variables enteros x e y
void Intercambio (int &a, int &b)
{
   int aux = a;      // almacenar valor original de x
   a = b;            // reemplazar x por y
   b = aux;          // asignar y al valor original de x
}
```

12.3.2. Análisis del algoritmo de ordenación por selección

El análisis del algoritmo de selección es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño de la lista o vector (array) y no de la distribución inicial de los datos. En la primera pasada se hacen *n-1* comparaciones, en la segunda pasada *n-2* y así sucesivamente. Matemáticamente se puede decir que en la pasada *i*, el número de comparaciones con la sublista `A[i+1]` a `A[n-1]` es

$$(n - 1) - (i + 1) + 1 = n - i - 1$$

de modo que el número total de comparaciones es

$$\sum_{0}^{n-2} (n-1) - i = (n-1)^2 - \sum_{i=0}^{n-2} i$$

$$= (n-1)^2 - (n-1)(n-2)/2$$

$$= \frac{1}{2} n (\mathrm{n} - 1)$$

o dicho de otra manera

$$(n-1) + (n-2) + (n-3) + ... + 1 = \frac{n(n-1)}{2}$$

Es decir, para ordenar un vector de *n* elementos, el número de comparaciones se calcula sumando los *n-1* primeros enteros

$$\frac{n\,(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

La complejidad del algoritmo se mide por el número de comparaciones y es *cuadrática*, es decir $0(n^2)$; el número de intercambios es $0(n)$. No hay caso mejor ni peor dado que el algoritmo realiza un número fijo de pasadas y explora o rastrea un número especificado de elementos en cada pasada.

12.4. ORDENACIÓN POR INSERCIÓN

El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista o archivo que ya está ordenado. Así, el proceso en el caso de la lista de enteros `A = 50, 20, 40, 80, 30`.

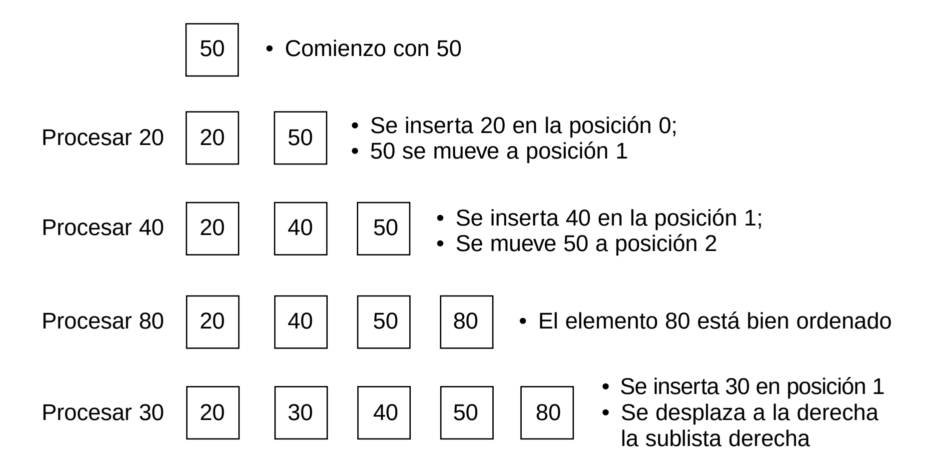


Figura 12.1. Método de ordenación por selección.

La función `OrdInsercion` tiene dos parámetros, el array `A` y el tamaño de la lista *n*. El algoritmo correspondiente contempla los siguientes pasos:

1. El primer elemento `A[0]` se considera ordenado; es decir, la lista inicial consta de un elemento.
2. Se inserta `A[1]` en la posición correcta; delante o detrás de `A[0]`, dependiendo de que sea menor o mayor. Es decir, se explora la lista desde `A[i]` hasta `A[n]` buscando la posición correcta de destino; esto es, la posición a insertar dentro de la lista ordenada.
3. Por cada bucle o iteración se mueve hacia abajo (a la derecha en la lista) una posición todos los elementos mayores que la posición a insertar, para dejar vacía esa posición.
4. Insertar el elemento a la posición correcta.

Ejemplo 12.4

Codificación del método de ordenación por inserción lineal.

La sublista `0 .. i – 1` *está ordenada, y se coloca el elemento que ocupa la posición* `i` *de la lista en la posición que le corresponde mediante una búsqueda lineal. De esta forma, la sublista se ordena entre la posición* `0 .. i`*. Si* `i` *varía desde* `1` *hasta* `n – 1` *se ordena la lista de datos.*

```
void Insercionlineal(float A[], int n)
{
   int i,j;
   bool encontrado;
   float auxiliar;

   for (i = 1; i < n; i++)
   {                             // A[0], A[1], ..., A[i-1] esta ordenado
     auxiliar = A[i];
```

```
      j = i - 1;
      encontrado = false;
      while (( j >= 0 ) && ! encontrado)
       if  (A[j] > auxiliar)
        {             // se mueve dato hacia la derecha para la insercion
            A[j + 1] = A[j];
            j--;
        }
        else
             encontrado = true;
      A[j + 1] = auxiliar;
    }
}
```

12.4.1. Algoritmo de inserción

```
// ordenar por inserción, ordenar las sublistas
// A[0]...A[i], 1 <= i <= n-1
// para cada i, se inserta A[i] en la posición correcta A[j]


template <class T>
void OrdInsercion (T A[], int n)
{
   int i, j;
   T aux;


   // i identifica la sublista A[0] a A[i]
   for (i = 1; i < n; i++)
   {
      // índice, j explora la lista desde A[i] buscando la
       // posición correcta del elemento destino, lo asigna a A[j]
      j = i;
      aux = A[i];
      // se localiza el punto de inserción explorando hacia abajo
      // mientras que aux < A[j-1] y no se haya encontrado el
      // principio de la lista

      while (j > 0 && aux < A[j-1])
      {
         // desplazar elementos hacia arriba para hacer espacio
         // para inserción
         A[j] = A[j-1];
         j--;
      }
      A[j] = aux;
   }
}
```

12.4.2. Análisis del algoritmo de ordenación por inserción

La ordenación por inserción requiere un número fijo de pasadas. Para una pasada general *i*, la inserción ocurre en la sublista `A[0]` a `A[i]` y requiere la media de i/2 comparaciones. El número total de comparaciones es

$$1/2 + 2/2 + 3/2 + ... + (n-2)/2 + (n-1)/2 = \frac{n(n-1)}{4}$$

Al contrario que otros métodos, la ordenación por inserción no utiliza intercambios. La complejidad del algoritmo es $0(n^2)$, que mide el número de comparaciones. El mejor caso sucede cuando la lista original está ya ordenada; en la pasada *i*, la inserción ocurre en `A[i]` y el número total de comparaciones es *1* con complejidad *0(n)*. El caso peor se produce cuando la lista está ordenada en orden descendente (inverso) y el número total de comparaciones es

$$(n-1) + (n-2) + ... + 3 + 2 + 1 = \frac{(n-1)n}{2}$$

Es decir, la complejidad es $0(n^2)$. Este método se mejora utilizando una *búsqueda binaria* con el propósito de encontrar la posición correcta para `A[i]` en la lista ordenada `A[0], ...A[i-1]`. Esta operación reduce la cantidad total de comparaciones de $0(n^2)$ a *0(nlogn)*. Sin embargo, incluso si la posición correcta se encuentra en *0(nlogn)* pasos, cada uno de los elementos `A[j+1] ... A[i-1]` debe moverse una posición y esta operación requiere $0(n^2)$ sustituciones. Desafortunadamente, aunque la técnica de búsqueda binaria mejora algo el algoritmo, no compensa la mejora en tiempo de ejecución la complejidad que se introduce en el algoritmo.

12.4.3. Codificación del método de ordenación por inserción binaria.

La única diferencia de los dos métodos de ordenación por inserción, radica en el método de realizar la búsqueda. En este caso es una búsqueda binaria, por lo que el desplazamiento de datos hacia la derecha debe hacerse después de haber encontrado la posición donde se debe insertar el elemento, que en esta codificación es siempre `izquierda`.

```
void Insercionbinaria(float A[], int n )
{
   int i, j, izquierda, derecha, centro;
   float auxiliar;

   for(i = 1; i < n; i++)
    {
        auxiliar = A[i];
        izquierda = 0;
        derecha = i - 1;
        while (izquierda <= derecha    //búsqueda binaria sin interruptor
        {
             centro = ( izquierda +derecha) / 2;
             if(A[centro] >  auxiliar)
                derecha = centro - 1;
             else
                izquierda = centro + 1;
        }
                                           // desplazar datos hacia la derecha
```

```
            for(j = i - 1; j >= izquierda; j--)
                    A[j + 1] = A[j];
            A[izquierda] = auxiliar;
          }
   }
```

12.2.4. Estudio de la complejidad de la inserción binaria

Los métodos de ordenación por inserción lineal e inserción binaria, sólo se diferencian entre sí, conceptualmente, en el método de la búsqueda. Por tanto, el número de movimientos de claves será el mismo en ambos casos es $\frac{n\,(n-1)}{2}$. En cuanto al número de comparaciones, cuando el intervalo tiene *i* elementos, se realizan $\log_2(i)$ comparaciones. Por tanto, el número de comparaciones es:

$$C = \sum_{i=1}^{n-1} \log_2(i) = \int_{1}^{n-1} \log_2(x)dx \backepsilon n \log_2(n)$$

Es decir, el número de comparaciones $O(n\log_2(n))$, y el número de movimientos es $O(n^2)$.

12.5. ORDENACIÓN POR BURBUJA

El método de *ordenación por burbuja* es el más conocido y popular entre estudiantes y aprendices de programación, por su facilidad de comprender y programar; por el contrario, es el menos eficiente y, por ello, normalmente, se aprende su técnica pero no suele utilizarse.

La técnica utilizada se denomina ***ordenación por burbuja*** u ***ordenación por hundimiento*** debido a que los valores más pequeños *«burbujean»* gradualmente (suben) hacia la cima o parte superior del array de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del array. La técnica consiste en hacer varias pasadas a través del array. En cada pasada se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están. Si una pareja está en orden decreciente, sus valores se intercambian en el array.

12.5.1. Algoritmo de la burbuja

En el caso de un array (lista) con *n* elementos, la ordenación por burbuja requiere hasta *n-1* pasadas. Por cada pasada se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo. Al final de cada pasada, el elemento mayor ha *«burbujeado»* hasta la cima de la sublista actual. Por ejemplo, después que la pasada 0 está completa, la cola de la lista *A[n-1]* está ordenada y el frente de la lista permanece desordenado. Las etapas del algoritmo son:

- En la pasada 0 se comparan elementos adyacentes.
  `(A[0],A[1]),(A[1],A[2]),(A[2],A[3]),...(A[n-2],A[n-1])`
  Se realizan *n*-1 comparaciones.
  Por cada pareja `(A[i] , A[i+1])` se intercambian los valores si `A[i+1] < A[i].`
  Al final de la pasada, el elemento mayor de la lista está situado en `A[n-1].`
- En la pasada 1 se realizan las mismas comparaciones e intercambios, terminando con el elemento de segundo mayor valor en `A[n-2].`
- El proceso termina con la pasada *n-1*, en la que el elemento más pequeño se almacena en `A[0].`

Un ejemplo ilustrará la técnica. Sea la lista:

```
a[ ]      25  60  45  35  12  92  85  30
```

En la primera pasada se hacen las siguientes comparaciones:

`a[0]`	con	`a[1]`	(25 con 60)	*no hay intercambio*
`a[1]`	con	`a[2]`	(60 con 45)	*intercambio*
`a[2]`	con	`a[3]`	(60 con 35)	*intercambio*
`a[3]`	con	`a[4]`	(60 con 12)	*intercambio*
`a[4]`	con	`a[5]`	(60 con 92)	*no hay intercambio*
`a[5]`	con	`a[6]`	(92 con 85)	*intercambio*
`a[6]`	con	`a[7]`	(92 con 30)	*intercambio*

Después de la primera pasada la lista está en el orden

```
25  45  35  12  60  85  30  92
```

Realizando las mismas comparaciones después de la segunda pasada la lista está en el orden

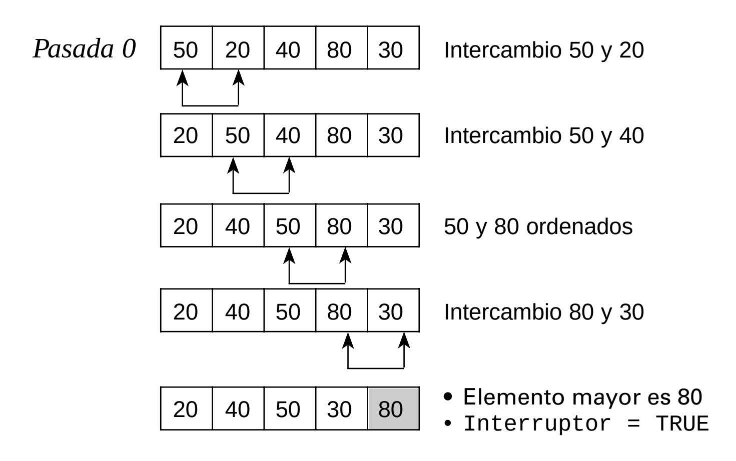
```
25  35  12  45  60  30  85  92
```

El conjunto completo de pasadas produciría:

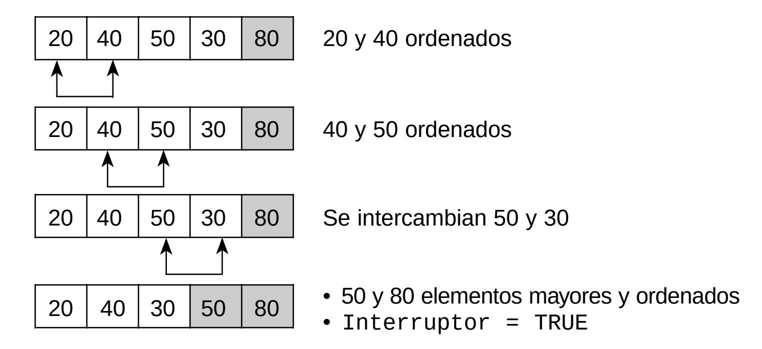
Pasada 0 (lista original)	25	60	45	35	12	92	85	30
Pasada 1	25	45	35	12	60	85	30	92
Pasada 2	25	35	12	45	60	30	85	92
Pasada 3	25	12	35	45	30	60	85	92
Pasada 4	12	25	35	30	45	60	85	92
Pasada 5	12	25	30	35	45	60	85	92
Pasada 6	12	25	30	35	45	60	85	92
Pasada 7	12	25	30	35	45	60	85	92

Obsérvese que la lista se ha quedado ordenada después de 5 *pasadas* (o *iteraciones*) por lo que las dos últimas pasadas son innecesarias y han consumido tiempo de ejecución. Es decir, el método de la burbuja admite alguna mejora en forma de eliminación de pasadas innecesarias. Con el objeto de eliminar esas pasadas innecesarias será preciso detectar cuándo la lista está ya ordenada. Esta situación es fácil de detectar ya que se producirá cuando no se haya producido ningún intercambio; este hecho se puede verificar mediante una *variable bandera* de tipo interruptor (verdadero/falso) que cambia de valor (estado) cuando se produce un intercambio.

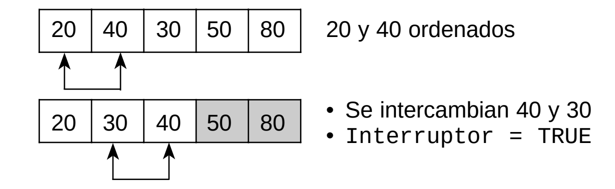
El ejemplo siguiente ilustra el funcionamiento del algoritmo de la burbuja con un array de 5 elementos (A = 50, 20, 40, 80, 30) donde se introduce una variable Interruptor para detectar si se ha producido intercambio en la pasada.



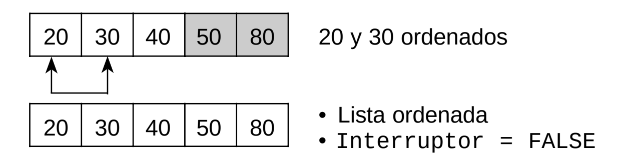
En la pasada 1



En la pasada 2, sólo se hacen comparaciones



En la pasada 3, se hace una única comparación de `20` y `30`, y no se produce intercambio.



En consecuencia, el algoritmo de ordenación de burbuja mejorado contempla dos bucles anidados: el *bucle externo* controla la cantidad de pasadas (al principio de la primera pasada todavía no se ha producido ningún intercambio, por tanto, la variable `interruptor` se pone a valor falso —`FALSE`—); el *bucle interno* controla cada pasada individualmente y cuando se produce un intercambio, cambia el valor de `interruptor` a verdadero (`TRUE`).

El algoritmo terminará, bien cuando se termine la última pasada (*n-1*) o bien cuando el valor del `interruptor` sea falso (`FALSE`), es decir, no se haya hecho ningún intercambio. Esta condición se define en la expresión lógica.

```
pasada < n-1 && interruptor = = TRUE. El algoritmo es:

void OrdBurbuja (int a[], int n)
{
   int aux, j, pasada;
   int interruptor = TRUE;
     for (pasada = 0; pasada < n-1 && interruptor == TRUE; Pasada++)
      {
         // bucle externo controla la cantidad de pasadas
         interruptor = FALSE; //no se ha hecho intercambio
                                   //todavía
         for (j = 0; j < n-pasada-1; j++)
            // bucle interno controla cada pasada
            if (a[j] > a[j+1])
```

```
            {
               // elementos desordenados
               // es necesario intercambio
               interruptor = TRUE;
               aux = a[j];
               a[j] = a[j+1];
               a[j+1] = aux;
            }  // fin de if
         }     // fin de for
}   // fin de OrdBurbuja
```

Una modificación al algoritmo anterior podría ser utilizar, en lugar de una variable `interruptor` con valor lógico, una variable `IndiceIntercambio` de tipo contador que se inicie a 0 (cero) al principio de cada pasada y se incremente en 1 cada vez que se produce un intercambio, de modo que cuando al terminar la pasada el valor de `IndiceIntercambio` siga siendo 0 implicará que no se ha producido ningún intercambio y, por consiguiente, la lista estará ordenada.

```
// Ordenación por burbuja: array A de n elementos
// Se realizan una serie de pasadas
// mientras IndiceIntercambio > 0

template < class T >
void OrdBurbuja2 (T a[], int n)
{
   int i, j;
   // Indice de último intercambio
   int IndiceIntercambio;

   // i es el índice del último elemento de la sublista
   i = n-1;

   // el proceso continúa hasta que no haya intercambios
   while (i > 0)
   {
      // se inicializa IndiceIntercambio a 0
      IndiceIntercambio = 0;

      // explorar la sublista a[0] a a[i]
      for (j = 0; j < i; j++)
         // intercambiar pareja y actualizar IndiceIntercambio
         if (a[j+1] < a[j])
         {
            intercambio (a[j], a[j+1]);
            IndiceIntercambio = j;
         }
      // i se pone al valor del índice del último intercambio
      // continúa ordenación de la sublista a[0] a a[i]
      i = IndiceIntercambio;
   }
}
```

12.5.2. Análisis del algoritmo de la burbuja

¿Cuál es la eficiencia del algoritmo de ordenación de la burbuja? Dependerá de la versión utilizada. En la versión más simple se hacen *n-1* pasadas y *n-1* comparaciones en cada pasada. Por consiguiente, el número de comparaciones es $(n-1) * (n-1) = n2 - 2n + 1$, es decir, la complejidad es $0(n^2)$.

Si se tienen en cuenta las versiones mejoradas haciendo uso de las variables `Interruptor` o `IndiceIntercambio`, entonces se tendrá una eficiencia diferente a cada algoritmo. En el mejor de los casos, la ordenación de burbuja hace una sola pasada en el caso de una lista que ya está ordenada en orden ascendente y, por tanto, su complejidad es $0(n)$. En el peor de los casos se requieren (n-i-1) comparaciones y (n-i-1) intercambios. La ordenación completa requiere $\frac{n\,(n-1)}{2}$ comparaciones y un número similar de intercambios. La complejidad para el peor de los casos es $0(n^2)$ comparaciones y $0(n^2)$ intercambios.

En cualquier forma, el análisis del caso general es complicado dado que alguna de las pasadas pueden no realizarse. Se podría señalar, entonces, que el número medio de pasadas K sea $0(n)$ y el número total de comparaciones es $0(n^2)$. En el mejor de los casos, la ordenación por burbuja puede terminar en menos de *n-1* pasadas pero requiere, normalmente, muchos más intercambios que la ordenación por selección y su prestación media es mucho más lenta, sobre todo cuando los arrays a ordenar son grandes.

12.6. ORDENACIÓN SHELL

La ordenación Shell se suele denominr también *ordenación por inserción con incrementos decrecientes*. Es una mejora de los métodos de inserción directa y burbuja, en el que se comparan elementos que pueden ser no contiguos. La idea general de algoritmo es la siguiente. Se divide la lista original (`n` elementos) en `n/2` grupos de dos elementos, con un intervalo entre los elementos de cada grupo de `n/2` y se clasifica cada grupo por separado (se comparan las parejas de elementos y si no están ordenados se intercambian entre sí de posiciones). Se divide ahora la lista en `n/4` grupos de cuatro con un intervalo o salto de `n/4` y, de nuevo, se clasifica cada grupo por separado. Se repite el proceso hasta que, en el último paso, se clasifica el grupo de `n` elementos. En el último paso el método Shell coincide con el método de la burbuja.

Ejemplo 12.5

Codificación del método de ordenación Shell.

```
void shell(float A[]. int n)
{
  int i, j, salto = n/2;
  float auxiliar;

  while (salto > 0)                     // ordenación de salto listas
  {
   for (i = salto; i < n; i++)          // ordenación parcial de lista i
   {                // los elementos de cada lista están a distancia salto
          j = i - salto;
           while(j >= 0)
           {
               k = j + salto;
               if (A[j] <= A[k]         // elementos contiguos de la lista
                   j = -1;                   // fin bucle par ordenado
```

```
                    else
                    {
                          auxiliar = A[j];
                          A[j] = A[k];
                          A[k] = auxiliar;
                          j = j - salto;
                    }
             }
     }
     salto = salto / 2;
    }
  }
```

Ejemplo 12.6

Seguimiento del método de ordenación de shell para la siguiente lista de datos 9, 18, 7, 9, 1, 7.

El resultado de ejecución de la función anterior, escribiendo los intercambios realizados así como el valor de salto en cada uno de ellos es:

```
  Vector desordenado
   9  18   7   9   1   7
  comienza  shell               cambio salto
   9   1   7   9  18   7         18  1   3
   1   9   7   9  18   7          9  1   1
   1   7   9   9  18   7          9  7   1
   1   7   9   9   7  18         18  7   1
   1   7   9   7   9  18          9  7   1
   1   7   7   9   9  18          9  7   1
  Vector ordenado
   1   7   7   9   9  18
Presione una tecla para continuar . . . _
```

12.7. BÚSQUEDA EN LISTAS: BÚSQUEDA SECUENCIAL Y BINARIA

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arrays y registros y, por ello, será necesario determinar si un array contiene un valor que coincida con un cierto *valor clave*. El proceso de encontrar un elemento específico de un array se denomina *búsqueda*. En esta sección se examinarán dos técnicas de búsqueda: *búsqueda lineal* o *secuencial*, la técnica más sencilla, y *búsqueda binaria* o *dicotómica*, la técnica más eficiente.

12.7.1. Búsqueda secuencial

La búsqueda secuencial busca un elemento de una lista utilizando un valor destino llamado ***clave***. En una búsqueda secuencial (a veces llamada ***búsqueda lineal***), los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro. La búsqueda secuencial es necesaria, por ejemplo, si se desea encontrar la persona cuyo número de teléfono es 958-220000 en un directorio o listado telefónico de su ciudad. Los directorios de teléfonos están organizados alfabéticamente por el nombre del abonado en lugar de por números de teléfono, de modo que deben explorarse todos los números, uno

después de otro, esperando encontrar el número 958-220000. Naturalmente, si se hace esta tarea se perderá con facilidad el número de modo manual, no así mediante la computadora que realizará fácilmente tareas repetitivas.

Naturalmente, existe otro método de búsqueda cuando se busca el número de teléfono del abonado señor Mackoy. Dado que el par *nombre de abonado/número de teléfono* está almacenado alfabéticamente por nombre, es posible buscar un nombre de modo eficiente; este método es el conocido como *búsqueda binaria* que aprovecha la ventaja de que los datos almacenados están ordenados. Éste es el sistema usual de búsqueda de un teléfono en un listín telefónico: búsqueda mediante aproximaciones sucesivas de la página donde aparece el nombre deseado.

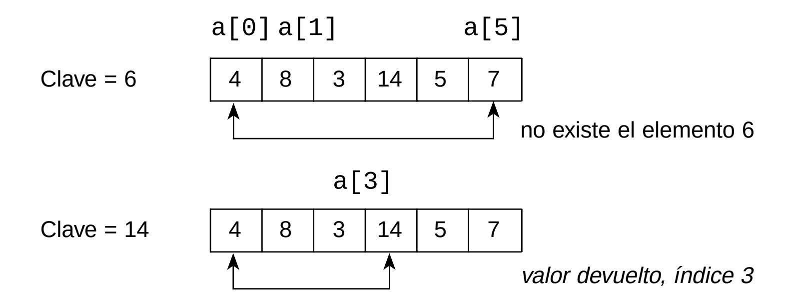
El algoritmo de búsqueda secuencial compara cada elemento del array con la *clave* de búsqueda. Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primero, el último o cualquier otro. De promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. El método de búsqueda lineal funcionará bien con arrays pequeños o no ordenados.

Algoritmo de búsqueda secuencial

El algoritmo comienza en el primer elemento `lista(0)` o bien en una posición predeterminada (*inicio*) y recorre los restantes elementos de las listas, comparando cada elemento con la clave. La exploración continúa hasta que se encuentra la clave o se termina la lista. Si la clave se encuentra, se devuelve el índice del elemento encontrado en la lista; en caso contrario, el valor –1.

Ejemplo 12.7

Buscar si el elemento 6 o el 14 están en la lista `A (4 8 3 14 5 7)`



El algoritmo en C++ es el siguiente:

```
int busquedaLineal (int lista [], int n, int clave)
{
   for (int i = 0; i < n; i++)
      if (lista[i] == clave)
         return i;

       return -i
}
```

Si por el contrario, se deseara comenzar la lista en un elemento distinto del cero, se declara un parámetro `inicio` que corresponde al elemento cuyo índice es `inicio;` es decir, `a[inicio]`. El algoritmo correspondiente sería:

```
int BusquedaSecuencial (int lista[], int inicio, int n, int clave)
```

```
{
   for (int i = inicio; i < n; i++)
      if (lista[i] == clave)
         return i;

      return -1;
}
```

Ejemplo 12.8

El programa comprueba la búsqueda secuencial contando el número de ocurrencias de una clave en una lista. El programa principal introduce 10 enteros en un array y, a continuación, solicita una clave.

El programa realiza repetidas llamadas a `BusquedaSec` utilizando un índice inicial diferente. Inicialmente, se comienza en el índice 0, el principio del array. Después de cada llamada a `BusquedaSec`, la cuenta del número de ocurrencias se incrementa si se encuentra la clave; en caso contrario la búsqueda termina y se saca la cuenta. Si se encuentra la clave, el valor devuelto identifica su posición en la lista. La siguiente llamada a `BusquedaSec` se hace con el valor inicial del índice de búsqueda en el elemento inmediatamente a la derecha; este artificio evitará repetir la búsqueda desde el principio y, por consiguiente, hacer más eficiente el algoritmo.

```
#include <iostream>
using namespace std;

// buscar el elemento clave en el array de enteros
// devuelve un puntero al dato o a NULL si se encuentra la clave
int BusquedaSec (int lista[], int inicio, int n, int clave)
{
   for (int i = inicio; i < n; i++)
      if (lista[i] == clave)
         return i; // devuelve el índice del elemento buscado
   return -1;     // búsqueda fallada, devuelve -1
}

void main (void)
{
   int a[10];
   int clave, cuenta = 0, pos;

   // introducción de una lista de 10 enteros
   cout << "Introduzca una lista de 10 enteros:" ;
   for (pos = 0; pos < 10; pos++)
      cin >> a[pos];

   cout << "Introduzca clave a buscar:";
   cin >> clave;
   // inicio búsqueda en el primer elemento del array
   pos = 0;

   // moverse a través de la lista hasta que se encuentre la
   // clave
   while ((pos = BusquedaSec(a, pos, 10, clave))!= -1)
   {
```

```
         cuenta ++;
         // avanzar al siguiente entero después de encontrar clave;
         pos++
      }

      cout << clave << " se repite " << cuenta
          << (cuenta-!= 1-? "veces"-:"vez")
          << " en la lista " << endl-;
   }
```

Al ejecutar el programa se visualizaría la siguiente salida:

```
Introduzca una lista de 10 enteros: 4 3 7 1 2 6 3 8 3 4
Introduzca clave a buscar: 3
3 se repite 2 veces en la lista
```

12.7.2. Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Una búsqueda binaria típica es la búsqueda de un número en un directorio telefónico o de un nombre en un diccionario. Dado el nombre, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra del primer apellido o de la palabra que busca. Se puede tener suerte y acertar con la página correcta; pero, normalmente, no será así y se mueve el lector a la página anterior o posterior del libro. Por ejemplo, si el nombre de la persona comienza con «J» y se está en la «L» se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que el nombre no está en la lista.

Una idea similar se aplica en la búsqueda en una lista ordenada. Se sitúa la lectura en el centro de la lista y se comprueba si nuestra clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sitúa uno en la mitad inferior o superior del elemento central de la lista. En general, si los datos de la lista están ordenados se puede utilizar esa información para acortar el tiempo de búsqueda.

Ejemplo 12.9

Se desea buscar a ver si el elemento 225 se encuentra en el conjunto de datos siguiente:

```
a[0]   a[1]   a[2]   a[3]   a[4]   a[5]   a[6]   a[7]
 13     44     75    100    120    275    325    510
```

El punto central de la lista es el elemento `a[3]` (100). El valor que se busca es 225 que es mayor que 100; por consiguiente, la búsqueda continúa en la mitad superior del conjunto de datos de la lista; es decir, en la sublista

```
a[4]   a[5]   a[6]   a[7]
 120    275    325    510
```

Ahora el último valor de la primera mitad de esta secuencia es 275; por consiguiente, el valor debe estar localizado en la secuencia

```
a[4]   a[5]
 120    275
```

El último valor de la primera mitad que buscamos es 120, que es más pequeño que el valor que se está buscando, de modo que se busca en la segunda mitad

```
a[5]
 275
```

Se observa, por último, que no ha habido éxito en la búsqueda ya que 225 < > 275.

Algoritmo de búsqueda binaria

Suponiendo que la lista está almacenada como un array, donde los índices de la lista son `bajo = 0` y `alto = n-1`, donde *n* es el número de elementos del array.

1. Calcular el índice del punto central del array

   `central = (bajo + alto)/2`

2. Comparar el valor de este elemento central con la clave

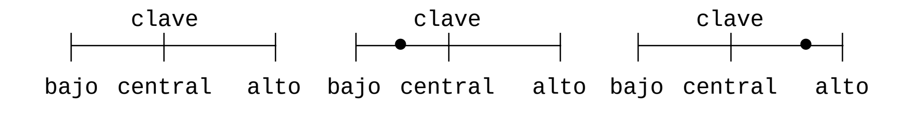
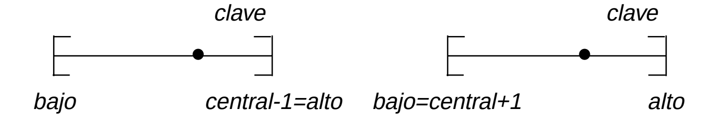


Figura 12.2. Búsqueda binaria de un elemento.

- Si `a[central] < clave`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo = central+1 ..alto.`
- Si `clave < a[central]`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo..central-1 = alto.`

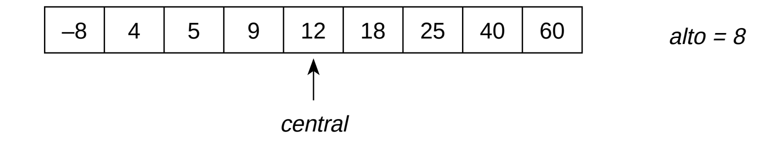


El algoritmo se termina bien porque se ha encontrado la clave o porque el valor de bajo excede a `alto` y el algoritmo devuelve el indicador de fallo de –1 (búsqueda no encontrada).

Ejemplo 12.10

Sea el array de enteros `A (-8, 4, 5, 9, 12, 18, 25, 40, 60)`, *buscar la clave,* clave = *40.*

1. `a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]   bajo = 0`



$$Central = \frac{bajo + alto}{2} = \frac{0 + 8}{2} = 4$$

`Clave (`**`40`**`) > a[4] (`**`12`**`)`

2. Buscar en sublista derecha

18	25	40	60

```
bajo = 5
alto = 8
```

$$Central = \frac{bajo + alto}{2} = \frac{5 + 8}{2} = 6{,}5{,}6$$

`Clave (`**`40`**`) > a[6] (`**`25`**`)`

3. Buscar en sublista derecha

40	60

```
alto = 8
bajo = 7
```

$$Central = \frac{bajo + alto}{2} = \frac{7 + 8}{2} = 7{,}5{,}7$$

`Clave (`**`40`**`) = a[7]   (40)` *búsqueda con éxito*

4. El algoritmo ha requerido (3) comparaciones frente a 8 comparaciones *(n-1, 9-1 = 8)* que se hubieran realizado con la búsqueda secuencial.

El código C++ del algoritmo de búsqueda binaria es

```
// búsqueda binaria
// devuelve el índice del elemento buscado o -1 caso de fallo

int BusquedaBin (TipoDato lista[], int bajo, int alto, Tipodato clave)
{
   int central;
   TipoDato valorcentral;

   while (bajo <= alto)
   {
      central = (bajo + alto)2        // índice de elemento central
      valorCentral = lista[central];  // valor del índice central
      if (clave == valorCentral)
          return central;             // encontrado valor;
                                      // devuelve posición
      else if (clave < valorCentral)
         alto = central -1            // ir a sublista inferior
      else
         bajo = central + 1;          // ir a sublista superior
   }                                  // elemento no encontrado
   return -1;
}
```

12.8. ANÁLISIS DE LOS ALGORITMOS DE BÚSQUEDA

Al igual que sucede con las operaciones de ordenación cuando se realizan operaciones de búsqueda, es preciso considerar la eficiencia (complejidad) de los algoritmos empleados en la búsqueda. El grado de eficiencia en una búsqueda será vital cuando se trata de localizar una información en una lista o tabla en memoria, o bien en un archivo de datos.

12.8.1. Complejidad de la búsqueda secuencial

La complejidad de la búsqueda secuencial diferencia entre el comportamiento en el peor y mejor caso. El mejor caso se encuentra cuando aparece una coincidencia en el primer elemento de la lista y en ese caso el tiempo de ejecución es $O(1)$. El caso peor se produce cuando el elemento no está en la lista o se encuentra al final de la lista. Esto requiere buscar en todos los n términos, lo que implica una complejidad de $O(n)$.

El caso medio requiere un poco de razonamiento probabilista. Para el caso de una lista aleatoria, es probable que una coincidencia ocurra en cualquier posición. Después de la ejecución de un número grande de búsquedas, la posición media para una coincidencia es el elemento central $n/2$. El elemento central ocurre después de $n/2$ comparaciones, que define el coste esperado de la búsqueda. Por esta razón, se dice que la prestación media de la búsqueda secuencial es $O(n)$.

12.8.2. Análisis de la búsqueda binaria

El caso mejor se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso la complejidad es $O(1)$ dado que sólo se realiza una prueba de comparación de igualdad. La complejidad del caso peor es $O(log_2n)$, que se produce cuando el elemento no está en la lista o el elemento se encuentra en la última comparación. Se puede deducir intuitivamente esta complejidad. El caso peor se produce cuando se debe continuar con listas de una longitud de 1. Cada iteración que falla debe continuar disminuyendo la longitud de la sublista por un factor de 2. El tamaño de las sublistas es:

$n \quad n/2 \quad n/4 \quad n/8 \; ... \quad 1$

La división de sublistas requiere m iteraciones, en donde m es aproximadamente log_2n. Para el caso peor, se dispone de una comparación inicial con el punto central de la lista y después una serie de log_2n iteraciones. Cada iteración requiere una operación de comparación:

Total comparaciones ∍ $1 + log_2n$

Como resultado, el caso peor de la búsqueda binaria es $O(log_2n)$. Este resultado se puede probar empíricamente. La relación (*ratio*) de los tiempos de ejecución para la búsqueda secuencial y binaria es 49,0. La relación teórica de los tiempos de ejecución esperados es aproximadamente 500/ ($log_2 1000$) = 50,2.

En [Ford, Topp, 1996][3] se considera un análisis formal de la búsqueda binaria que conduce a la complejidad ya citada $O(log_2n)$. Este análisis es el siguiente:

- La primera iteración del bucle trata con la lista completa
- Cada iteración posterior tiene tamaños de las sublistas que se van dividiendo sucesivamente por 2.

$n \quad n/2 \quad n/2^2 \quad n/2^3 \quad n/2^4 \; .... \; n/2^m$

---

[3] William Ford y William Topp, *Data Structures with C++*, *Englewood Cliffs*, New Jersey-, Prentice Hall, 1996, páginas 194-196.

Eventualmente, *m* será un entero tal que

$n/2^m < 2 \quad$ *o bien* $\quad n < 2^{m+1}$

Dado que *m* es el primer entero para el cual $n/2^m < 2$, debe ser verdad que

$n/2^{m-1} \ \forall 2 \quad$ *o bien* $\quad 2m \ \Im \ n$

Por consiguiente,

$2^m \ \Im \ n < 2^{m+1}$

Tomando logaritmos en base 2 en la expresión anterior quedará

$m \ \Im \ log_2 n = x < m+1$

En síntesis, el caso medio es también $O(log_2 n)$.

12.8.3. Comparación de la búsqueda binaria y secuencial

La comparación en tiempo entre los algoritmos de búsqueda secuencial y binaria se va haciendo espectacular a medida que crece el tamaño de la lista de elementos. Tengamos presente que en el caso de la búsqueda secuencial en el peor de los casos coincidirá el número de elementos examinados con el número de elementos de la lista, tal como representa su complejidad *O(n)*.

Sin embargo, en el caso de la búsqueda binaria, tengamos presente, por ejemplo, que 2^{10} = 1.024, lo cual implica el examen de 11 posibles elementos; si se aumenta el número de elementos de una lista a 2.048 y teniendo presente que 2^{11} = 2.048 implicará que el número de elementos examinados en la búsqueda binaria es 11/12. Si se sigue este planteamiento, se puede encontrar el número *n* más pequeño, tal que

$2^n \ \forall 1.000.000$

Es decir, 2^{19} = 524.288, $\quad 2^{20}$ = 1.048.576

La Tabla 12.1 muestra la comparación de los métodos de búsqueda secuencial y búsqueda binaria. En la misma tabla se puede apreciar una comparación del número de elementos que se deben examinar utilizando búsqueda secuencial y binaria. Esta tabla muestra la eficiencia de la búsqueda binaria comparada con la búsqueda secuencial y cuyos resultados de tiempo vienen dados por las funciones de complejidad *O(logn)* y *O(n)* de las búsquedas binaria y secuencial respectivamente.

TABLA 12.1. Comparación de las búsquedas binaria y secuencial.

Números de elementos examinados		
Tamaño de la lista	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000

RESUMEN

- Una de las aplicaciones más frecuentes en programación es la ordenación.
- Los datos se pueden ordenar en orden ascendente o en orden descendente.
- Cada recorrido de los datos durante el proceso de ordenación se conoce como *pasada* o *iteración.*
- Los algoritmos de ordenación básicos son:
  - Selección.
  - Inserción.
  - Burbuja.
- Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos.*
- La eficiencia de los algoritmos de burbuja, inserción y selección es $0(n^2)$.
- La búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista.
- Existen dos métodos básicos de búsqueda en arrays: **búsqueda secuencial** y **binaria**.
- La **búsqueda secuencial** se utiliza normalmente cuando el array no está ordenado. Comienza en el principio del array y busca hasta que se encuentra el dato buscado o se llega al final de la lista.
- Si un array está ordenado, se puede utilizar un algoritmo más eficiente denominado **búsqueda binaria**.
- La eficiencia de una lista secuencial es $0(n)$.
- La eficiencia de una búsqueda binaria es $0(log_2 n)$.

EJERCICIOS

12.1. Eliminar todos los números duplicados de una lista o vector (array). Por ejemplo, si el array toma los valores

4 7 11 4 9 5
11 7 3 5

ha de cambiarse a

4 7 11 9 5 3

12.2. Escribir un programa que lea hasta un máximo de 10 letras y escriba de nuevo las letras en la pantalla en orden inverso. Por ejemplo, si la entrada es

`abcd`

la salida debe ser

`dcba`

(Utilice un punto como valor centinela que señale el final de la entrada.)

12.3. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, a trazar las etapas necesarias para encontrar el número 88.

8	13	17	26	44	56	88
97						

Igual búsqueda pero para el número 20.

Un array contiene los elementos siguientes:

3	13	7	26	44	23	98
57						

¿Cuántas pasadas serán necesarias para ordenar en orden ascendente por el método de burbuja, selección e inserción?

12.4. Escribir un programa que lea 10 nombres y los ponga en orden alfabético utilizando el método de selección.

12.5. Clasificar el vector

```
42    57    14    40    96
19    08    68
```

por los métodos: (*a*) burbuja; (*b*) selección; (*c*) inserción. Cada vez que se reorganiza el vector se debe mostrar el nuevo vector.

12.6. Escribir un programa de búsqueda lineal para un vector ordenado.

12.7. Supongamos que se tiene una secuencia de *n* números que deben ser clasificados:

1. Utilizar el método de selección, ¿cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si:
   - Ya está clasificado,
   - Está en orden inverso?
2. Repetir el paso 1 para el método de la burbuja.

PROBLEMAS

12.1. Modificar el algoritmo de ordenación por selección que ordene un vector de empleados por salario.

12.2. Escribir un programa de consulta de teléfonos. Leer un conjunto de datos de 1.000 nombres y números de teléfono de un archivo que contiene los números en orden aleatorio. Las consultas han de poder realizarse por nombre y por número de teléfono.

12.3. Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria. Una lista A se rellena con 2.000 enteros aleatorios en el rango 0 .. 1999 y, a continuación, se ordena. Una segunda lista B se rellena con 500 enteros aleatorios en el mismo rango. Los elementos de B se utilizan como claves de los algoritmos de búsqueda.

Nota: Las funciones de tiempo (TickCount) se definen en el archivo «tics.h» y devuelve el número de 1/60º de segundo desde el arranque del sistema. El tiempo se mide en función de las 500 búsquedas que se utilizan en cada algoritmo. La salida incluye el tiempo en segundos y el número de coincidencias.

12.4. Realizar un programa que permita la introducción de 12 números enteros en una lista y mediante un menú seleccione y ejecute uno de los siguientes métodos de ordenación: selección, intercambio, inserción y burbuja. El programa debe presentar en pantalla las listas ordenadas en orden creciente o decreciente a opción del usuario.

12.5. Construir una función que permita ordenar por fechas y de mayor a menor un vector de *n* elementos que contiene datos de contratos (n <= 50). Cada elemento del vector debe ser una estructura con los campos (entero) día, mes, año y número de contrato.

Nota: Pueden existir diversos contratos con la misma fecha, pero no números de contrato repetidos.

12.6. Escribir un programa que cree un array de 100 enteros aleatorios en el rango de 1 a 200 y, a continuación, utilizando una búsqueda secuencial, realizar la búsqueda de 50 enteros seleccionados aleatoriamente (iguales o distintos). Al final del programa se deben visualizar las siguientes estadísticas:

- Número de búsquedas con éxito
- Número de búsquedas fallidas
- Porcentajes de éxito y de fallo.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 255).

2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

12.1. ¿Cuál es la diferencia entre búsqueda y ordenación?

12.2. Un vector contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción. ¿Cómo estarán colocados los elementos del vector después de cuatro pasadas más del algoritmo?

3, 13, 8, 25, 45, 23, 98, 58.

12.3. ¿Cuál es la diferencia entre ordenación por selección y ordenación por inserción?

12.4. Dada la siguiente lista 47, 3, 21, 32, 56, 92. Después de dos pasadas de un algoritmo de ordenación, el array se ha quedado dispuesto así: 3, 21, 47, 32, 56, 92. ¿Qué algoritmo de ordenación se está utilizando (selección, burbuja o inserción)? Justifique la respuesta.

12.5. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88 y el número 20.

8, 13, 17, 26, 44, 56, 88, 97.

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 256).

2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

12.1. Modificar el algoritmo de ordenación por selección para que ordene un vector de enteros en orden descendente.

12.2. Escribir un programa que cree un array de 100 enteros aleatorios en el rango de 1 a 200 y, a continuación, utilizando una búsqueda secuencial, realizar la búsqueda de 50 enteros seleccionados aleatoriamente (iguales o distintos). Al final del programa se deben visualizar las siguientes estadísticas:

- Número de búsquedas con éxito.
- Número de búsquedas fallidas.
- Porcentajes de éxito y de fallo.

12.3. Escribir una mejora del método de ordenación de burbuja, consistente en no realizar más iteraciones cuando se detecte que el vector ya está ordenado.

12.4. Modificar el método de ordenación por selección, para que en lugar de colocar el elemento menor en el lugar que le corresponde, lo haga con el elemento mayor.

12.5. Escribir una función que acepte como parámetro un vector que puede contener elementos duplicados. La función debe sustituir cada valor repetido por –1 y devolver al punto donde fue llamado el vector modificado y el número de entradas modificadas (puede suponer que el vector de datos no contiene el valor –1 cuando se llama a la función).

12.6. Escribir un programa que genere un vector de 1.000 números enteros aleatorios positivos y menores que 100, y haga lo siguiente:

1. *Visualizar los números de la lista en orden creciente.*
2. *Calcular e imprimir la mediana (valor central).*
3. *Determinar el número que ocurre con mayor frecuencia.*
4. *Imprimir una lista que contenga*

   a) *números menores de 30,*
   b) *números mayores de 70,*
   c) *números que no pertenezcan a los dos grupos anteriores.*

12.7. Construir una función que permita ordenar por fechas y de mayor a menor un vector de n elementos que contiene datos de contratos (*n* <= 50). Cada elemento del vector debe ser una estructura con los campos (entero) día, mes, año y número de contrato.

12.8. Escribir un programa para leer un texto de datos formado por n cadenas de caracteres, siendo n una variable entera. La memoria que ocupa el texto se ha de ajustar al tamaño real. Una vez leído el texto se debe ordenar el texto original por la longitud de las líneas. El programa debe mostrar el texto leído y el texto ordenado.

PARTE II

PROGRAMACIÓN ORIENTADA A OBJETOS

CAPÍTULO 13

Clases y objetos

Contenido

INTRODUCCIÓN

Hasta ahora hemos aprendido el concepto de estructuras, con lo que se ha visto un medio para agrupar datos. También se han examinado funciones que sirven para realizar acciones determinadas a las que se les asigna un nombre. En este capítulo se tratarán las clases, un nuevo tipo de dato cuyas variables serán objetos. Una **clase** es un tipo de dato que contiene código (funciones) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, tal como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En el capítulo se aprenderá a crear (definir y especificar) y utilizar clases individuales y en capítulos posteriores se verá cómo definir y utilizar jerarquías y otras relaciones entre clases.

CONCEPTOS CLAVE

- Análisis y diseño orientado a objetos.
- Concepto de clase.
- Concepto de objeto.
- Constructores.
- Destructores.
- Encapsulamiento.
- Especificaciones de acceso `public`, `protected`, `private`.
- Funciones `const`.
- Funciones miembro.
- Miembros dato.
- Ocultación de la información.

13.1. CLASES Y OBJETOS

El paradigma orientado a objetos nació en 1969 de la mano del doctor noruego Kristin Nygaard que, intentando escribir un programa de computadora que describiera el movimiento de los barcos a través de un fiordo, descubrió que era muy difícil simular las mareas, los movimientos de los barcos y las formas de la línea de la costa con los métodos de programación existentes en ese momento. Descubrió que los elementos del entorno que trataba de modelar —barcos, mareas y línea de la costa de los fiordos— y las acciones que cada elemento podía ejecutar, constituían unas relaciones que eran más fáciles de manejar.

Las tecnologías orientadas a objetos han evolucionado mucho pero mantiene la razón de ser del paradigma: combinación de la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos como instancias o ejemplares de ellas son los elementos clave sobre los que se articula la orientación a objetos.

13.1.1. ¿Qué son objetos?

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos. Los objetos tienen propiedades específicas, tales como posición, tamaño, color, forma, textura, etc., que definen su estado. Los objetos también tienen ciertos comportamientos que los hacen diferentes de otros objetos.

Booch[1] define un *objeto* como «algo que tiene un estado, un comportamiento y una identidad». Supongamos una máquina de una fábrica. El *estado* de la *máquina* puede estar *funcionando/parando* («on/of»), su potencia, velocidad máxima, velocidad actual, temperatura, etc. Su *comportamiento* puede incluir acciones para arrancar y parar la máquina, obtener su temperatura, activar o desactivar otras máquinas, condiciones de señal de error o cambiar la velocidad. Su *identidad* se basa en el hecho de que cada instancia de una máquina es única, tal vez identificada por un número de serie. Las características que se eligen para enfatizar el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En un diseño de un programa orientado a objetos, se crea una abstracción (un modelo simplificado) de la máquina basado en las propiedades y comportamiento que son útiles en el tiempo.

Martin y Odell definen un objeto como «cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos». Para realizar esa actividad se añaden a cada objeto de la clase los propios datos asociados con sus propias funciones miembro que pertenecen a la clase.

Cualquier programa orientado a objetos puede manejar muchos objetos. Por ejemplo, un programa que maneja el inventario de un almacén de ventas al por menor, utiliza un objeto de cada producto manipulado en el almacén. El programa manipula los mismos datos de cada objeto, incluyendo el número de producto, descripción del producto, precio, número de artículos del *stock* y el momento de nuevos pedidos.

Cada objeto conoce también cómo ejecutar acciones con sus propios datos. El objeto producto del programa de inventario, por ejemplo, conoce cómo crearse a sí mismo y establecer los valores iniciales de todos sus datos, cómo modificar sus datos y cómo evaluar si hay artículos suficientes en el *stock* para cumplir una petición de compra. En esencia, la cosa más importante de un objeto es reconocer que consta de datos, y las acciones que pueden ejecutar.

Un objeto de un programa de computadora no es algo que se pueda tocar. Cuando un programa se ejecuta, la mayoría existen en memoria principal. Los objetos se crean por un programa para su uso mientras el programa se está ejecutando. A menos que se guarden los datos de un objeto en un disco, el objeto se pierde cuando el programa termina (este objeto se llama *transitorio* para diferenciarlo del objeto *permanente* que se mantiene después de la terminación del programa).

---

[1] Booch, Grady, *Análisis y diseño orientado a objetos con aplicaciones*, Madrid, Díaz de Santos/Addison-Wesley, 1995. Esta obra fue traducida al español por el autor de este libro y por el profesor Cueva Lovelle.

Un *mensaje* es una instrucción que se envía a un objeto y que cuando se recibe ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con el objeto mediante su *interfaz*, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. *Una interfaz* se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico tal como una máquina de fax tiene una interfaz de usuario bien definida; por ejemplo, esa interfaz incluye el mecanismo de avance del papel, botones de marcado, receptor y el botón «enviar». El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles. De hecho, la apertura de la máquina durante el período de garantía puede anularla.

13.1.2. ¿Qué son clases?

En términos prácticos, una *clase* es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch denomina a una clase como «un conjunto de objetos que comparten una estructura y comportamiento comunes».

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios*, *métodos* o *funciones miembro*. El término *función miembro* se utiliza, específicamente, en C++. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como *atributos* o *variables*. El término *atributo* se utiliza en análisis y diseño orientado a objetos y el término *variable* se suele utilizar en programas orientados a objetos.

Las clases son la espina dorsal de la mayoría de los programas C++. C++ soporta la programación orientada a objetos con un modelo de objetos basado en clases. Es decir, una clase define el comportamiento y el estado de los objetos que son instancias de la clase.

Las clases definen nuestros propios tipos de datos que son personalizados a los problemas a resolver, obteniendo aplicaciones que son más fáciles de escribir y de comprender. Los tipos de clases bien diseñadas pueden ser tan fáciles de utilizar como los tipos incorporados.

Una clase define miembros dato y funciones. Los miembros dato almacenan el estado asociado con los objetos del tipo de clase y las funciones ejecutan operaciones que dan significado a los datos. Las clases pueden separar la interfaz y la implementación. Solamente los implementadores de la clase necesitan conocer los detalles de la implementación; el usuario normalmente necesita conocer los detalles de la interfaz.

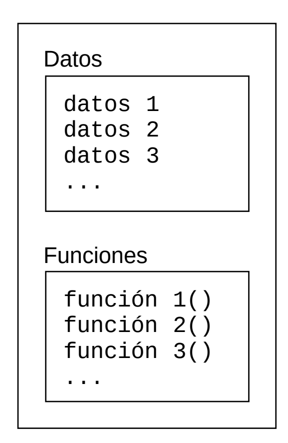


Figura 13. 1. Las clases contienen datos y funciones.

Los tipos de clases se conocen como tipos *abstractos de objetos*. Un TAD trata los datos (el estado) y las operaciones sobre el estado como única unidad: se precisa en modo abstracto lo que hace la clase en lugar de pensar en cómo funciona la clase internamente.

Una *clase* es un medio para traducir la abstracción de un tipo definido por el usuario. Combina la representación de los datos miembros (*atributos*) y las funciones miembro (*métodos*) que manipulan esos datos en una única entidad o paquete que se trata como tal. La colocación de datos y funciones juntas en una sola entidad (*clase*) es una idea central en programación orientada a objetos. La Figura 13.1 muestra la representación gráfica de una clase.

13.2 DEFINICIÓN DE UNA CLASE

La primera operación que se debe realizar con una clase es definirla. La *definición* o *especificación* tiene dos partes:

- *Declaración de la clase*. Describe el componente datos, en términos de *miembros dato* y la interfaz pública, en términos de funciones miembro, también denominados métodos.
- *Definiciones de métodos de las clases*. Describen la implementación de las funciones miembro.

La declaración de una clase utiliza la palabra reservada `clase`, seguida de la lista de miembros dato y métodos de la clase, encerrados entre llaves.

Sintaxis

```
class NombreClase                // Identificador válido
{
  declaraciones de datos         // atributos
  declaraciones de funciones     // métodos
}; <-------- Punto y coma obligatorio
```

Regla práctica

- Los atributos (*datos*) son variables simples (de tipo entero, estructuras, arrays, `float`, etc.).
- Los métodos (*funciones*) son funciones simples que actuan sobre los atributos (*datos*).
- La omisión del punto y coma detrás de la llave de cierre es un error frecuente, difícil de detectar.

En una clase, por convenio de denominación, los miembros están ocultos al exterior; es decir, los datos y funciones miembro son *privados por* omisión. La *visibilidad* permite controlar el acceso a los miembros de la clase, ya que sólo se puede acceder a los miembros que son visibles desde el exterior y queda prohibido el acceso a aquellos miembros ocultos. Los especificadores o duplicadores de acceso son: `public` (*público*), `protected` y `private` (*privado*) (Tabla 13.1).

Tabla 13.1. Secciones pública y privada de una clase.

Sección	Comentario
`public` (*público*)*	Se permite el acceso desde el exterior del objeto de la clase (visible desde el exterior)
`private` (*privado*)**	Se permite el acceso sólo desde el interior del objeto (oculto al exterior)

* A los miembros públicos se puede acceder desde cualquier objeto de la clase.
** A los miembros privados sólo se puede acceder desde métodos de la propia clase.

> ### *Nota*
>
> Si la clase se define con la palabra reservada `struct` todos los miembros de la estructura son visibles desde el exterior con el objeto de mantener la compatibilidad con C. Ésta es la única diferencia existente en C++ entre palabras reservadas `class` y `struct`.

La sintaxis completa de una clase es:

```
class NombreClave
{
   public:
      Sección pública   // declaración de miembros públicos
   private:
      Sección pivada    // declaración de miembros privado
};
```

Esta característica de la clase es una propiedad de la programación orientada a objetos: la *ocultación de datos*. El mecanismo principal para conseguir la ocultación de datos es ponerlos todos en una clase y hacerlos privados. A los datos o funciones privados sólo se puede acceder desde el interior de la clase y a los datos y funciones públicas se puede acceder desde el exterior de la clase. La ocultación de datos es una técnica de programación que facilita a los programadores el diseño evitándoles errores que en programación estructurada suele ser frecuentes. La Figura 13.2 muestra gráficamente el funcionamiento de las secciones pública y privada de una clase.

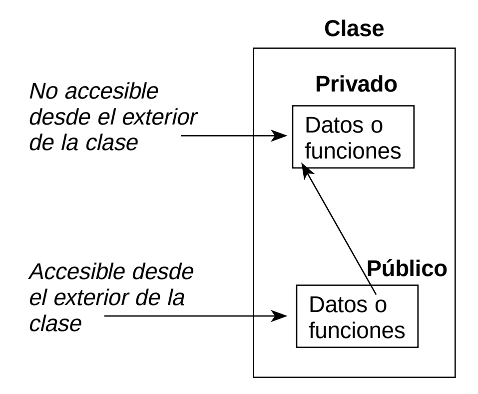


Figura 13.2. Secciones pública y privada de una clase.

Ejemplo

Declaración de una clase `Semaforo`

```
clase Semaforo
{
   public:
      void cambiarColor();
      //...
   private:
      enum Color {VERDE, ROJO, AMBAR};
      Color c;
};
```

En la clase `Semaforo` se observa que la sección pública declara la definición de los servicios (acción de cambio de color del semáforo) y en la sección privada —en último lugar— los detalles internos del semáforo (diferentes tipos de colores del semáforo).

Reglas prácticas en la declaración de una clase

1. Las declaraciones (cabeceras de las funciones) de los métodos, normalmente, se colocan en la sección pública y las declaraciones de los datos (atributos), normalmente, se situan en la sección privada.
2. Es indiferente colocar primero la sección pública o la sección privada; pero la situación más usual es colocar la sección pública primero para resaltar las operaciones que forman parte de la interfaz pública del usuario.
3. Las palabras claves `public` y `private` seguidas de dos puntos, señalan el comienzo de las respectivas secciones públicas y privadas; aunque no se suele hacer, una clase puede tener varias secciones pública y privada.
4. *Recuerde* poner el punto y coma después de la llave de cierre de la declaración de la clase.

¿Control de acceso a los miembros: Público o Privado?

Los miembros de una clase se pueden declarar públicos o privados. Para ello, el cuerpo de una clase contiene dos palabras clave reservadas: `private` y `public`; tanto los datos como las funciones miembro se pueden declarar en una u otra sección.

Sin embargo, uno de los principios fundamentales de la programación orientada a objetos es el *encapsulamiento (ocultación de la información)*; por esta razón, normalmente, los datos deben permanecer ocultos y por ello se deben poner en la sección privada. Por el contrario, las funciones miembro que constituyen la interfaz de la clase van en la sección pública, ya que en caso contrario no se podría llamar a estas funciones desde un programa. Normalmente, se utilizan funciones miembro privadas para manejar detalles de la implementación que no forman parte de la interfaz pública.

Ejemplo

Diversas clases con diferentes secciones

```
class DemoUno
{
   private
      float precio;
      char nombre[20];
   public:
   int calanlar(int)
   ...
};
```

```
class DiaDelAnio
{
public:
   void leer();
   void escribir();
   int mes;
   int dia;
};
```

No se necesita utilizar la palabra reservado `private` en las declaraciones de la clase ya que el control de acceso por defecto para los objetos de la clase es privado. Sin embargo, con frecuencia utilizaremos la etiqueta `private` con el objetivo de enfalizar el concepto de ocultación de datos.

Ejemplo

```
class DemoUno
{
   float precio;
```

```
   char nombre[20];
public:
   vid calcular(int)
   ...
};
```

Las funciones son públicas y los datos privados

Normalmente, los datos dentro de una clase son privados y las funciones son públicas. Esto se debe al modo en que se usan las clases. Los datos están ocultos para asegurarse frente a manipulaciones accidentales, mientras que las funciones que operan sobre los datos son públicos de modo que se pueda acceder a ellas desde el exterior de la clase. Sin embargo, no es una regla estricta y en algunas circunstancias se puede necesitar utilizar funciones privadas y datos públicos.

```
                                  Palabra reservada
                                  Nombre de la clase
         class Prueba
       ┌►{
       │    private: ──────────── Palabra reservada private
       │       int datos;  }      funciones y datos privados
       │       ...
llaves │    public: ───────────── Palabra reservada public
       │    void funcdemo(int d)  }  funciones y datos públicas
       │       {datos = d;}
       └►};
          └──────── punto y coma
```

Para utilizar una clave se necesitan tres cosas:

1. ¿Cuál es su nombre?
2. ¿Dónde está definida?
3. ¿Qué operaciones soporta?

El nombre de la clase, por ejemplo, `Semaforo`, está definida en un archivo de cabecera, que al igual que otros archivos tiene dos partes: un nombre de archivo y el sufijo (extensión) del archivo. Normalmente el nombre del archivo de la clase es el mismo que el nombre de la clase. La extensión, normalmente, es `.h`, pero algunos compiladores y programas pueden usar `.H`, `.hpp` o `.hxx`. En el ejemplo anterior, el archivo se denomina `Semaforo.h`.

Excepto en algunos casos especiales que se verán más adelante, en la declaración de la clase sólo se colocan las cabeceras de los métodos. Sus definiciones se colocan, normalmente, en un archivo independiente llamado *archivo de implementación* cuyo nombre es el del archivo de cabecera correspondiente, pero su extensión es `.cpp` en lugar de `.h`. Por ejemplo, `Semaforo.cpp`. Los dos archivos constituyen una biblioteca de clases y los programas que utilizan la clase definida en la biblioteca se denominarán *programas cliente*. Estos programas deben incluir la cabecera de la biblioteca con la directiva del compilador:

```
#include "NombreClase.h"
#include "Semaforo.h"
```

Nota

Observe la diferencia entre

```
#include <iostream>
```

e

```
#include "Semaforo.h"
```

El primer caso indica al compilador de C++ que `iostream` es una biblioteca estándar; y el segundo formato con el nombre del archivo entre comillas, es una biblioteca definida por el programador.

Ejemplo 13.1

Definición de una clase llamada `Punto` *que contiene las coordenadas x e y de un punto en un plano.*

```
class Punto {
public:
   int Leerx();            // devuelve el valor de x
   void Fijarx(int);       // establece el valor de x

private:
   int x;                  // coordenada x
   int y;                  // coordenada y
};
```

La definición de una clase no reserva espacio en memoria. El almacenamiento se asigna cuando se crea un objeto de una clase (*instancia* de una clase). Las palabras reservadas `public` y `private` se llaman *especificadores de acceso*.

Ejemplo 13.2

Declaración de la clase `edad` *(almacenada en el archivo* `edad.h`*)*

```
class edad
{
   private:
      int edadHijo, edadMadre, edadPadre;      //datos
   public:
      edad();
      void iniciar (int, int, int);            //función miembro
      int leerHijo();
      int leerMadre();
      int leerPadre();
};
```

Una *declaración* de una clase consta de una palabra reservada `class` y el nombre de la clase. Una declaración de la clase se utiliza cuando el compilador necesita conocer una determinada clase definida totalmente en alguna parte del programa. Por ejemplo,

```
class Punto;      // definida en algún lugar
```

13.2.1. Objetos de clases

Una vez que una clase ha sido definida, un programa puede contener una *instancia* de la clase, denominada un *objeto de la clase*.

> ***Formato***
>
> ```
> nombre_clase   identificador ;
> ```

Así, la definición de un objeto `Punto` es:

```
Punto P;
```

Cada clase define un tipo. El nombre del tipo es el mismo que el nombre de la clase. Por consiguiente, la clase `Semaforo` define un tipo denominado `Semaforo`. Al igual que sucede con los tipos de datos incorporados (propios) del lenguaje, se puede definir una variable de un tipo:

```
Semaforo S1;
```

en donde se indica que `S1` es un objeto del tipo `Semaforo`.
Un programa que puede utilizar datos del objeto Semaforo puede ser:

```
#include <iostream>
using namespace std

#include "Semaforo.h"
int main ()
{
   Semaforo S1;
   ...
   std :: cin >> S1;
   ...
   std :: cont << S1 << endl;
   return 0;
}
```

Un objeto tiene la misma relación con su clave que una variable tiene con un tipo de dato. Un objeto se dice que es una *instancia* (un *ejemplar*) de una clase, de igual modo que un Seat Ibiza es una instancia de un vehículo.

El valor de una variable de un tipo clase se llama *objeto* (en general, cuando se hable de una variable de un tipo clase, normalmente nos estamos refiriendo a un objeto). Un objeto tiene miembros función y miembros dato. En programación con clases, un programa se visualiza como una colección de objetos que interactúan. Los objetos pueden interactuar ya que son capaces de acciones, es decir, invocaciones de funciones miembro. Las variables de un tipo clase se declaran de igual modo que las variables de tipos predefinidos y de igual modo que las variables estructura.

> La definición de objetos significa *crear* esos objetos. Este proceso se llama también *instanciación*. El término *instanciar* (del inglés *instantiate*) se debe a que se crea una instancia (un ejemplar, un caso específico de una clase. Los objetos se denominan, a veces, *variables* de instancia (*instance variables*).

El *operador de acceso* a un miembro (`.`) selecciona un miembro individual de un objeto de la clase. Las siguientes sentencias, por ejemplo, crean un punto `P`, que fija su coordenada `x` y la visualiza a continuación.

```
Punto p;
p.Fijarx (100);
cout << " coordenada   x es " << p.Leerx();
```

El operador punto se utiliza con los nombres de las funciones miembro para especificar que son miembros de un objeto.

Ejemplo

```
clases DiaSemana;      // contiene una función Visualizar
   DiaSemana Hoy;      // Hoy es un objeto
   Hoy.Visualizar();   // ejecuta la función Visualizar
```

Se puede asignar un objeto de una clase a otro; por defecto, C++ realiza una copia bit a bit de todos los miembros dato. En otras palabras, todos los miembros físicamente contenidos en el área de datos del objeto fuente se copian en el objeto receptor. Por ejemplo, el siguiente código crea un punto (`Punto`) llamado `P2` y lo inicializa con el contenido de `P`:

```
Punto P;
// ...
Punto P2;
P2 = P;
```

13.2.2. Acceso a miembros de la clase: encapsulamiento

Un principio fundamental en programación orientada a objetos es la *ocultación de la información* que significa que a determinados datos del interior de una clase no se puede acceder por funciones externas a la clase. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o funciones privados sólo se puede acceder desde dentro de la clase. Por el contrario, los datos o funciones públicos son accesibles desde el exterior de la clase.

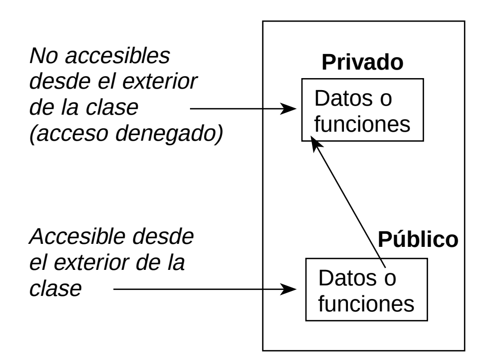


Figura 13.3. Secciones pública y privada de una clase.

Se utilizan tres diferentes *especificadores de acceso* para controlar el acceso a los miembros de la clase son: `public`, `private` y `protected`. Se utiliza el siguiente formato general en definiciones de

la clase, situando primero las funciones miembro públicas, seguidas por los miembros protegido y privado (este orden, sin embargo, no es obligatorio).

Formato

```
class nombre_clase {
public:
   // miembros públicos
protected:
   // miembros protegidos
private:
   // miembros privados
};
```

El especificador `public` define miembros públicos, que son aquellos a los que se puede acceder por cualquier función. A los miembros que siguen al especificador `private` sólo se puede acceder por funciones miembro de la misma clase o por funciones y clases amigas[2]. A los miembros que siguen al especificador `protected` se puede acceder por funciones miembro de la misma clase o de clases derivadas de la misma, así como por amigas. Los especificadores `public, protected` y `private` pueden aparecer en cualquier orden.

En la Tabla 13.2 cada «x» indica que el acceso está permitido al tipo del miembro de la clase listado en la columna de la izquierda.

Tabla 13.2. Visibilidad.

Tipo de miembro	Miembro de la misma clase	Amiga	Miembro de una clase derivada	Función no miembro
`private`	x	x		
`protected`	x	x	x	
`public`	x	x	x	x

Si se omite el especificador de acceso, el acceso por defecto es privado. En la siguiente clase `Estudiante`, por ejemplo, todos los datos son privados, mientras que las funciones miembro son públicas.

```
class Estudiante {
   long numId;
   char nombre[40];
   int edad;

public:
   long LeerNumId();
   char * LeerNombre();
   int LeerEdad();
};
```

---

[2] Las funciones y clases amigas se estudian más adelante.

El mismo especificador de acceso puede aparecer más de una vez en una definición de una clase, pero —en este caso— no es fácil de leer.

```
class Estudiante{
   private:
      long numId;
   public:
      long LeerNumId();
   private:
      char nombre[40];
      int edad;
   public:
      char * LeerNombre();
      int LeerEdad();
};
```

El especificador de acceso se aplica a *todos* los miembros que vienen después de él en la definición de la clase (hasta que se encuentra otro especificador de acceso).

Aunque las secciones públicas y privadas pueden aparecer en cualquier orden, en C++, los programadores suelen seguir algunas reglas en el diseño que citamos a continuación, y de entre las cuales puede elegir la que considere más eficiente.

1. Poner la sección privada primero, debido a que contiene los atributos (datos).
2. Poner la sección pública primero debido a que las funciones miembro y los constructores son la interfaz del usuario de la clase.

La regla 2 presenta realmente la ventaja de que los datos son algo secundario en el uso de la clase y con una clase definida adecuadamente, realmente no se suele necesitar nunca ver cómo están declarados los atributos.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El principio de ocultación de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida que permite que los detalles de implementación de los objetos sean ignorados. Por consiguiente, las funciones miembro y los miembros datos de la *sección pública* forman la interfaz externa del objeto, mientras que los elementos de la *sección privada* son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.

El principio de *encapsulamiento* significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

Nota

C++ proporciona un especificador de acceso, `protected`. La explicación detallada de este tercer especificador se explicará en el Capítulo 14 ya que se requiere el conocimiento del concepto de herencia.

13.2.3. Datos miembros (Miembros dato)

El concepto de **miembro de datos** o **dato miembro** es el atributo del objeto. Los miembros dato pueden ser de cualquier tipo válido definido, con excepción del propio tipo de la clase que se está definiendo, ya que está incompleto; es posible, sin embargo, emplear punteros o referencias al tipo.

Los miembros dato se pueden declarar constantes. Si un miembro dato es `const` se indica que su valor, establecido durante la constitución del objeto, no puede ser modificado con posterioridad.

Los datos de una clase pueden existir en cualquier número, al igual que pueden existir cualquier número de elementos dato en una estructura. Normalmente, los datos de una clase son privados y van situados a continuación de la palabra reservada `private`, de modo que se pueden acceder a ellos desde el interior de la clase pero no desde el exterior.

Las *funciones miembro* son funciones que están incluidas dentro de una clase (en algunos lenguajes de programación como Smalltalk, a estas funciones se les denomina *métodos* y por analogía, a veces, en C++ también se les llama *métodos*).

La definición de una función miembro debe incluir el nombre de la clase ya que puede darse el caso de que dos o más claves tengan funciones miembro con el mismo nombre. La clase

```
class DiaDeSemana {
public:
   void salida ();
   int semana;
   int dia;
};
```

En un programa puede existir una sola clase como DiaDeSemana, pero en otros programas puede haber muchas definiciones de clases y más de una clase puede tener funciones miembro con el mismo nombre. La definición de una función miembro es similar a la definición de una función ordinaria excepto que se debe especificar el nombre de la clase en la cabecera de la definición de la función. Así en el caso de la función `salida`, la cabecera de su definición puede ser:

```
void DiaDeSemana :: salida ()
```

Las funciones miembro se definen de modo similar a las funciones ordinarias. Al igual que cualquier otra función, una función miembro consta de cuatro partes:

- Un tipo de retorno de la función.
- El nombre de la función.
- Una lista de parámetros (puede ser vacía) separadas por comas.
- El cuerpo de la función que está contenido entre una pareja de símbolos llave (`{}`).

Las tres primeras partes constituyen el prototipo de la función. Este prototipo define toda la información de los tipos relativos a la función: su tipo de retorno, nombre de la función y el tipo de argumentos que se pueden pasar a la misma. El prototipo de la función *debe estar definido* dentro del cuerpo de la clase. El cuerpo de la clase, sin embargo, puede estar definido dentro de la propia clase o fuera del cuerpo de la clase.

Ejemplo

Declaración de una clave `Articulo_Ventas`

```
class Articulo_Ventas {
public:
```

```
   // operaciones sobre objetos de la clase
   double precio_medio() const;
   bool articulo_igual (const Articulo_Ventas & art) const
      {return iva == art.iva; }
   // miembros privados
private:
   // ...
};
```

Se deben declarar todos los miembros de una clase dentro de las llaves que delimitan la definición de la clase. No existe ningún medio para añadir miembros a la clase. Los miembros que son funciones deben estar definidos y declarados. Se puede definir una función miembro dentro o fuera de la clase. En el ejemplo de la clave `Articulo_Ventas`, la función `articulo_igual` se define dentro de la clase, mientras que la función `articulo_igual` se declara dentro de la clase pero se define en otro sitio.

Una función miembro que se declara dentro de una clase se considera implícitamente como función en línea. Es posible *declarar* una función dentro de una clase pero *definirla* en cualquier otro sitio. Las funciones definidas fuera de la clase no están, normalmente, en línea.

Ejemplo

```
class DemoFunc
{public:
   void leerdatos (int d)
      {datosprueba = d; }
   void verdatos ()
      {cunt << "\n los datos son: " << datosprueba;}
};
```

Obsérvese que las funciones miembro `leerdatos()` y `verdatos()` son *definiciones* en las cuales el código real de la función está contenido en la definición de la clase. (Recuerde que las funciones no se definen al estilo de las variables que reservan espacio de memoria automáticamente.)

Definición de una función miembro fuera de la clase

Las funciones miembro definidas fuera de la definición de la clase deben indicar que son miembros de la clase. En este caso, como ya se ha comentado, la función se declara dentro de la clase con una sintaxis similar a:

```
tipo nombre (parámetros);
```

Este prototipo indica al compilador que la función es un miembro de la clase pero que se definirá fuera de la declaración de la clase, en alguna parte del listado. La sintaxis de la definición de la función requiere el operador de resolución de ámbito (::) y es la siguiente:

```
tipo_devuelto Nombre_clase :: Nombre_función (lista_parámetros)
{
   sentencias del cuerpo de la función
}
Lista_par lista de declaración de parámetros
```

Ejemplo

```
void DiaDeSemana :: Salida ()
{
   switch (semana)
   {
      case 1:
         cout << "Lunes"; break;
      case 2:
         cout << "Martes"; break;
      ...
      case 7:
         cout << "Domingo"; break;
      default:
         cout << "Error en DiaDeSemana -- ";
   }
   cout << dia;
}
```

13.2.4. Funciones miembro

En la Figura 13.4 se muestra la definición completa de una clase.

```
class Producto  <------------ Nombre de la clase
{                             Acceso para almacenamiento de datos
   private:                   declaraciones para almacenamiento
      int num_prod;           de datos
      char nombre_prod[30]-; descrip_pred[80];
      float precio_prod, num_unidades;
   public:  <----------- Acceso para funciones
      Producto(int, char[], char[], float, float);
      void verProducto();
      float obtenerPrecio();
      void actualizarProd(int); <------ Declaración de funciones
};
```

Figura 13.4. Definición típica de una clase.

Ejemplo 13.3

La clase Punto *define las coordenadas de un punto en un plano. Por cada dato se proporciona una función miembro que devuelve su valor y una que fija su valor*

```
class Punto {
public:
   int LeerX() { return x; }
   int LeerY() { return y; }
   void FijarX (int valx) { x = valx; }
   void FijarY (int valy) { y = valy; }
```

```
private:
   int x;
   int y;
};
```

Ejemplo 13.4

```
void DiaAnyo:: visualizar()
{
    cout  << "mes   = " << mes
          << ", dia = " << dia << endl;
}
```

Las declaraciones de las funciones en este ejemplo son también definiciones ya que se ha incluido el cuerpo de cada función. Si un cuerpo de la función consta de una única sentencia, muchos programadores sitúan el cuerpo de la función en la misma línea que el nombre de la función. Por otra parte, si una función contiene múltiples sentencias, cada una debe ir en una línea independiente. Por ejemplo:

```
class Punto {
public:
   void FijarX(int valx)
   {
      if ((valx >= -100) && (valx <= 100))
         x = valx;
      else
         cerr << “Error-: Fijarx() argumento fuera de rango”;
   }
   // ...
};
```

Ejercicio

Definir una clase que represente una radio que emita con tecnología MP3.

```
class radioMP3 {
private:
   int frecuencia;
   int volumen;
public:
   void iniciar (void);
   void AumentarFrecuencia (void);
   void DisminuirFrecuencia (void);
   void BajarVolumen (void);
   void SubirVolumen (void);
};

void radioMP3 :: iniciar (void)
{
   // inicializar atributos de frecuencia y volumen
   frecuencia = 99.99;
```

```
   volumen = 45;
}

void radioMP3 :: SubirVolumen (void)
{
   // Incrementar en una unidad el volumen
   volumen++;
}

void radioMP3 :: BajarVolumen (void)
{
   // Disminuir en una unidad el volumen
   volumen --;
}

void radioMP3 :: DisminuirFrecuencia (void)
{
   // disminuir frecuencia 1 MHz
   frecuencia--;
{

void radioMP3 :: AumentarFrecuencia (void)
{
   // aumentar la frecuencia 1 MHz
   frecuencia++;
}
```

Un programa `main` que utilice las funciones de `radioMP3` podría ser:

```
int main (void)
{
   radioMP3 miEstacion;

   miEstacion.iniciar();
   miEstacion.SubirVolumen();
   miEstacion.BajarVolumen();
   miEstacion.AumentarFrecuencia();
   miEstacion.DisminuirFrecuencia();
}
```

12.2.5. Llamadas a funciones miembro

Las funciones miembro de una clase se invocan o llaman mediante el operador punto (`.`) con la siguiente sintaxis:

`nombreObjeto.nombreFunción (valores de los parámetros)`

Ejemplo

```
class Demo
{
```

```
private:
   // ...
public
   void func1 (int P1)
      {...}
   void func2 (int P2)
      {...}
};

Demo d1, d2;      // definicion de los objetos d1 y d2
...
d1.func (2005);
d2.func (2010);
```

La función `func1()` es una función miembro de la clase `Demo` y se debe llamar siempre en conexión con un objeto de esa clase. No tiene sentido una llamada directa `func1(2005)`; y produce un error. Una función miembro se llama para actuar sobre un objeto específico y no sobre la clase en general. Para utilizar una función miembro, el operador punto (`.`) (denominado operador de acceso a miembros de una clase) conecta el nombre del objeto y la función miembro.

Regla

A las funciones miembro de una clase sólo se puede acceder por un objeto de esa clase.

Mensajes

Algunos lenguajes orientados a objetos denominan a las invocaciones a las funciones miembro como *mensajes*. Por consiguiente la llamada

```
d1.func1();
```

representa el *envío de un mensaje* al objeto `d1` para que ejecute una tarea determinada. El término *mensaje* no es un término formal en C++, pero es útil interpretar la tecnología orientadas a objetos. En el caso de C++ se enfatiza en que los objetos son entidades discretas y nos comunicamos con ellos llamando a sus funciones miembro.

Ejercicio

```
//demoObj.cpp
//demostracion del uso de objetos
#include <iostream>
using namespace std;

class demoObj {
private:
   int undato;                  //datos de la clase
public:
   void fijardatos (int d)      //establecer datos
      { un dato = d; }
```

```
   void mostrardatos ()      //visualizar datos
      { cout << "El dato es: " << un dato << endl;
};

int main ()
{
   demoObj d1, d2;

   d1.fijardatos (2005);
   d2.fijardatos (2010);

   d1.mostrardatos ();
   d2.mostrardatos ();
   return 0;
}
```

13.2.6. Tipos de funciones miembro

Las funciones miembro que pueden aparecer en la definición de una clase se clasifican en función del tipo de operación que representan.

- *Constructores* y *destructores*, son funciones miembro a las que se llama automáticamente cuando un objeto se crea o se destruye.
- *Selectores* que devuelven los valores de los miembros dato.
- *Modificadores* o *mutadores* que permiten a un programa cliente cambiar los contenidos de los miembros dato.
- *Operadores* que permiten definir operadores estándar C++ para los objetos de las clases.
- *Iteradores* que procesan colecciones de objetos, tales como arrays y listas.

13.2.7. Funciones en línea y fuera de línea

Hasta este momento, todas las funciones miembro se han definido dentro del cuerpo de la definición de la clase. Se denominan definiciones de funciones *en línea* (`inline`). Para el caso de funciones más grandes, es preferible codificar sólo el prototipo de la función dentro del bloque de la clase y codificar la implementación de la función en el exterior. Esta forma permite al creador de una clase ocultar la implementación de la función al usuario de la clase proporcionando sólo el código fuente del archivo de cabecera, junto con un archivo de implementación de la clase precompilada.

En el siguiente ejemplo, `FijarX` de la clase `Punto` se declara pero no se define en la definición de la clase:

```
class Punto {
public:
   void FijarX(int valx);
private:
   int x;
   int y;
};
```

Ejercicio 13.1

Definir una clase `DiaAnyo` que contiene los atributos `mes` y `día` y una función miembro `Visualizar`. El mes se registra como un valor entero en `mes` (1, Enero, 2, Febrero, etc.). El día del mes se registra en la variable entera `día`. Escribir un programa que haga uso de la clase y ver su salida.

```
// Uso de la clase DiaAnyo
#include <iostream>
using namespace std;

class DiaAnyo
{
   public:
      void visualizar();
      int mes;
      int dia;
};
// programa que usa DiaAnyo
int main()
{
   DiaAnyo hoy, cumpleanyos;

   cout << "Introduzca fecha del dia de hoy\n";
   cout << "Introduzca el numero del mes:";
   cin  >> hoy.mes;
   cout << “Introduzca el dia del mes: ";
   cin  >> hoy.dia;
   cout << " Introduzca su fecha de nacimiento:\n";
   cout << " Introduzca el numero del mes:";
   cin  >> cumpleanyos.dia;
   cout << "Introduzca el dia del mes: ";
   cin  >> cumpleanyos.mes;

   cout << " La fecha de hoy es ";
   hoy.visualizar(); // llamada a la función visualizar
   cout << " su fecha de nacimiento es ";
   cumpleanyos.visualizar();

   if(hoy.mes == cumpleanyos.mes
      && hoy.dia == cumpleanyos.dia)
      cout << "¡Feliz cumpleaños! \n";
   else
      cout << "¡Feliz día! \n";
   return 1
}
```

Ejecución del programa

```
Introduzca fecha del día de hoy:
Introduzca el número del mes : 08
Introduzca el día del mes :10
Introduzca su fecha de nacimiento:
```

```
Introduzca el número del mes : 2
Introduzca el día del mes :25
La fecha de hoy es mes = 8, día = 10
Su fecha de nacimiento es mes = 2, día = 25
¡Feliz día!
```

La implementación de una función miembro externamente a la definición de la clase, se hace en una definición de la función *fuera de línea*. Su nombre debe ser precedido por el nombre de la clase y el signo de puntuación `::` denominado *operador de resolución de ámbito*. El operador `::` permite al compilador conocer que `FijarX` pertenece a la clase `Punto` y es, por consiguiente, diferente de una función global que pueda tener el mismo nombre o de una función que tenga ese nombre que puede existir en otra clase. La siguiente función global, por ejemplo, puede coexistir dentro del mismo ámbito que `Punto::FijarX` :

```
void FijarX(int valx)
{
   // ...
}
```

Formato

El símbolo `::` (*operador de resolución de ámbitos*) se utiliza en sentencias de ejecución que accede a los miembros estáticos de la clase. Por ejemplo, la expresión `Punto::X` se refiere al miembro dato estático `X` de la clase `Punto`.

13.2.8. La palabra reservada `inline`

La decisión de elegir funciones en línea y fuera de línea es una cuestión de eficiencia en tiempo de ejecución. Una función en línea se ejecuta normalmente más rápida, ya que el compilador inserta una copia «fresca» de la función en un programa en cada punto en que se llama a la función. La definición de una función miembro en línea no garantiza que el compilador lo haga realmente en línea; es una decisión que el compilador toma, basado en los tipos de las sentencias dentro de la función y cada compilador de C++ toma esta decisión de modo diferente.

Si una función se compila en línea, se ahorra tiempo de la UCP (CPU) al no tener que ejecutar una instrucción *«call»* (llamar) para bifurcar a la función y no tener que ejecutar una instrucción de *return* para retornar al programa llamador. Si una función es corta y se llama cientos de veces, se puede apreciar un incremento en eficiencia cuando actúa como función en línea.

Una función localizada fuera del bloque de la definición de una clase se puede beneficiar de las ventajas de las funciones en línea si está precedida por la palabra reservada `inline`:

```
inline void Punto::FijarX (int valx)
{
   x = valx;
}
```

Dependiendo de la implementación de su compilador, las funciones que utilizan la palabra reservada `inline` se puede situar en el mismo archivo de cabecera que la definición de la clase. Las funciones que no utilizan `inline` se sitúan en el mismo módulo de programa, pero no el archivo de cabecera. Estas funciones se sitúan en un archivo `.cpp` (`.cc`, `.cxx`, `.c`, etc.).

13.2.9. Nombres de parámetros de funciones miembro

Al igual que sucede con los prototipos de funciones globales, se pueden omitir los nombres de parámetros de una declaración de funciones miembro e identificar sólo los tipos de parámetros. Por ejemplo:

```
class Punto {
public:
   void FijarX(int);
   // ...
};
```

Sin embargo, esta situación no siempre es deseable, ya que como la definición de la clase es también la interfaz de la clase, una función miembro sin más información que los tipos de datos de parámetros no proporcionará información suficiente sobre cómo llamar a la función:

```
class Demo {
public :
   FuncionDemo(int, float, char * ,int);
   // ...
};
```

Regla

Si los nombres de los parámetros aparecen tanto en la declaración como en la implementación de la función, no es necesario que los nombres sean idénticos pero su orden y tipo si que han de serlo :

```
class Punto {
public:
   void Girar(int valx, int valy);
   // ...
};

void Punto::Girar(int x, int y)
{
   // ...
}
```

Consejo

Es conveniente incluir comentarios de una o dos líneas en la declaración de una función que indiquen lo que hace la función y cuáles son los valores de entrada/salida correspondientes.

13.2.10. Implementación de clases

El código fuente para la implementación de funciones miembro de una clase es código ejecutable. Se almacena, por consiguiente, en archivos de texto con extensiones `.cp` o `.cpp`. Normalmente, se sitúa la implementación de cada clase en un archivo independiente.

Cada implementación de una función tiene la misma estructura general. Obsérvese que una función comienza con una línea de cabecera que contiene, entre otras cosas, el nombre de la función y su cuerpo está acotado entre una pareja de signos llave.

Las clases pueden proceder de diferentes fuentes:

- Se pueden declarar e implementar sus clases propias. El código fuente siempre estará disponible.
- Se pueden utilizar clases que hayan sido escritas por otras personas o incluso que se han comprado. En este caso, se puede disponer del código fuente o estar limitado a utilizar el código objeto de la implementación.
- Se puede utilizar clases de las bibliotecas del programa que acompañan a su software de desarrollo C++. La implementación de estas clases se proporcionan normalmente como código objeto.

En cualquier forma, se debe disponer de las versiones de texto de las declaraciones de clase para que pueda utilizarlas su compilador.

13.2.11. Archivos de cabecera y de clases

Las declaraciones de clases se almacenan normalmente en sus propios archivos de código fuente, independientes de la implementación de sus funciones miembro. Éstos son los *archivos de cabecera* que se almacenan con una extensión `.h` o bien `.hpp` en el nombre del archivo. Aunque como ya conoce el lector, la versión estándar ANSI/ISO no requiere, normalmente, las extensiones `.h` ni `.hpp` en los nombres de los archivos de cabecera.

El uso de archivos de cabecera tiene un beneficio muy importante: «Se puede tener disponible la misma declaración de clases a muchos programas sin necesidad de duplicar la declaración». Esta propiedad facilita la reutilización en programas C++.

Para tener acceso a los contenidos de un archivo de cabecera, un archivo que contiene la implementación de la funciones de la clase declaradas en el archivo de cabecera o un archivo que crea objetos de la clase declarada en el archivo de cabecera *incluye* (`include`), o mezcla, el archivo de cabecera utili-

```
//Declaración  de una clase almacenada en Demo1.h
class Demo1
{
   public:
      Demo1();
      void Ejecutar();
};
```

```
// Declaración de la clase edad almacenada en edad.h
class edad
{
   private:
      int edadHijo, edadPadre, edadMadre;
   public:
      edad();
      void iniciar(int, int, int);
      int obtenerHijo();
      int obtenerPadre();
      int obtenerMadre();
};
```

Figura 13.5. Listado de declaraciones de clases.

zando una *directiva de compilador*, que es una instrucción al compilador que se procesa durante la compilación. Las directivas del compilador comienzan con el signo «almohadilla» (#).

La directiva que mezcla el contenido de un archivo de cabecera en un archivo que contiene el código fuente de una función es:

```
#include "nombre-archivo"
```

Opciones de compilación

La mayoría de los compiladores soporta dos versiones ligeramente diferentes de esta directiva. La primera instruye al compilador a que busque el archivo de cabecera en un directorio de disco que ha sido designado como el depósito de archivos de cabecera.

Ejemplo

```
#include <iostream>
     |
```

utiliza la biblioteca de clases que soporta E/S

La segunda versión se produce cuando el archivo de cabecera está en un directorio diferente; entonces, se pone el nombre del camino entre dobles comillas.

Ejemplo

```
#include "\mi_cabecera\cliente.h"
```

13.3. CONSTRUCTORES

Un *constructor* es una función miembro de propósito específico que se ejecuta automáticamente cuando se crea un objeto de una clase. Un constructor sirve para inicializar los miembros dato de una clase.

Un constructor tiene el mismo nombre que la propia clase. Cuando se define un constructor no se puede especificar un valor de retorno, ni incluso `void` (un constructor nunca devuelve un valor). Un constructor puede, sin embargo, tomar cualquier número de parámetros (cero o más).

Un objeto comienza a existir cuando se crea, y en un momento determinado deja también de existir. Normalmente, las funciones miembro de una clase se utilizan para dar valores a los elementos dato de un objeto. Sin embargo, es conveniente, a veces, que un objeto se pueda inicializar por sí mismo cuando se crea, sin necesidad de una llamada independiente a una función miembro. La inicialización automática se ejecuta utilizando una función miembro llamada constructor.

Un **constructor** es una función miembro que se ejecuta automáticamente siempre que se crea un objeto. Dicho de otro modo, en el momento que se crea un objeto se invoca automáticamente al constructor y en el momento que deja de existir se invoca a otra función miembro llamada *destructor*.

Las función constructor se utiliza normalmente para **inicializar** los atributos, mientras que la función destructor se suele utilizar para liberar memoria que haya sido reservada con anterioridad.

Reglas

1. El constructor tiene el mismo nombre que la clase.
2. Puede tener cero, o más parámetros.
3. No devuelve ningún valor.

Ejemplo 13.5

La clase `Rectángulo` *tiene un constructor con cuatro parámetros*

```
class Rectángulo
{
   private:
      int Izdo;
      int Superior;
      int Dcha;
      int Inferior;
   public:
      // Constructor
      Rectangulo(int I, int S, int D, int Inf);
      // definiciones de otras funciones miembro
};
```

Cuando se define un objeto, se pasan los valores de los parámetros al constructor utilizando una sintaxis similar a una llamada normal de la función:

```
Rectangulo Rect(25, 25, 75, 75);
```

Esta definición crea una instancia del objeto `Rectangulo` e invoca al constructor de la clase pasándole los parámetros con valores especificados.

Caso particular

Se puede también pasar los valores de los parámetros al constructor cuando se crea la instancia de una clase utilizando el operador `new`:

```
Rectangulo *Crect = new Rectangulo(25, 25, 75, 75);
```

El operador `new` invoca automáticamente al constructor del objeto que se crea (ésta es una ventaja importante de utilizar `new` en lugar de otros métodos de asignación de memoria tales como la función `malloc`).

13.3.1. Constructor por defecto

Un constructor que no tiene parámetros se llama *constructor por defecto*. Un constructor por defecto normalmente inicializa los miembros dato asignándoles valores por defecto.

Ejemplo 13.6

El constructor por defecto inicializa `x` *e* `y` *a 0.*

```
class Punto {
public:
   Punto()            // constructor
   {
      x = 0;
```

```
      y = 0;
   }
   private:
   int x;
   int y;
};
```

Una vez que se ha declarado un constructor, cuando se declara un objeto `Punto` sus miembros dato se iniciliazan a `0`. Ésta es una buena práctica de programación.

```
Punto P1;   // P1.x = 0, P1.y = 0
```

Si `Punto` se declara dentro de una función, su constructor se llama tan pronto como la ejecución del programa alcanza la declaración de `Punto`:

```
void FuncDemoConstructorD()
{
   Punto Pt;   // llamada al constructor
   // ...
}
```

Regla

C++ crea automáticamente un constructor por defecto cuando no existen otros constructores. Sin embargo, tal constructor no inicializa los miembros dato de la clase a un valor previsible, de modo que siempre es conveniente al crear su propio constructor por defecto, darle la opción de inicializar los miembros dato con valores previsibles.

Precaución

Tenga cuidado con la escritura de la siguiente sentencia:

```
Punto  P();
```

Aunque parece que se realiza una llamada al constructor por defecto, lo que se hace es declarar una función de nombre `P` que no tiene parámetros y devuelve un resultado de tipo `Punto`.

13.3.2. Constructores alternativos

Como ya se ha visto anteriormente es posible pasar argumentos a un constructor asignando valores específicos a cada miembro dato de los objetos de la clase. Un constructor con parámetros se denomina *constructor alternativo*.

Ejemplo

```
Punto P(50, 250);      // define e inicializa P
```

Un constructor se puede llamar directamente, creando un objeto temporal.

Ejemplo

```
Punto x = Punto(50, 250) ;
```
se construye un punto y se asigna a `x`

Ejemplo

Otro constructor alternativo de la clase `P`, que visualiza un mensaje después de ser llamado.

```
Punto::Punto(int valx, int valy)
{
   FijarX(valx);
   FijarY(valy);
   cout << "Llamado el constructor de Punto .\n";
}
```

13.3.3. Constructores sobrecargados

Al igual que se puede sobrecargar una función global, se puede también sobrecargar el constructor de la clase o cualquier otra función miembro de una clase excepto el destructor (posteriormente se describirá el concepto de destructor, pero no se puede sobrecargar). De hecho, los constructores sobrecargados son bastante frecuentes; proporcionan medios alternativos para inicializar objetos nuevos de una clase.

Sólo un constructor se ejecuta cuando se crea un objeto, con independencia de cuántos constructores hayan sido definidos.

Ejemplo 13.7

```
class Punto {
public:
   Punto();                     // constructor
   Punto(int valx, int valy);   // constructor sobrecargado
   // ...
private:
   int x;
   int y;
};
```

La declaración de un objeto `Punto` puede llamar a cualquier constructor

```
Punto P;           // llamada al constructor por defecto
Punto Q(25,50);    // llamada al constructor alternativo
```

13.3.4. Constructor de copia

Existe un tipo especializado de constructor denominado *constructor de copia*, que se crea automáticamente por el compilador. El constructor de copia se llama automáticamente cuando un objeto se pasa por valor: se construye una copia local del objeto que se construye. El constructor de copia se llama también cuando un objeto se declara e inicializa con otro objeto del mismo tipo.

Ejemplo

```
Punto P;
Punto T(P);
Punto Q = P;
```

Por defecto, C++ construye una copia bit a bit de un objeto. Sin embargo, se puede también implementar el constructor de la copia y se utiliza para notificar al usuario que una copia se ha realizado, normalmente como una ayuda de depuración.

Regla

Si existe un constructor alternativo, C++ no generará un constructor por defecto. Para prevenir a los usuarios de la clase de crear un objeto sin parámetros, se puede: 1) omitir el constructor por defecto; o bien, 2) hacer el constructor privado.

Ejemplo

```
class Punto {
public:
   Punto(int valx, int valy);
private:
   Punto();
   // ...
};
...
Punto T;  // error; el constructor no está accesible
```

13.3.5. Inicialización de miembros en constructores

No está permitido inicializar un miembro dato de una clase cuando se define. Por consiguiente, la siguiente definición de la clase genera errores:

```
class C {
private:
   int T = 0;                  // Error
   const int CInt = 25;        // Error
   int &Dint = T               // Error
// ...
};
```

No tiene sentido inicializar un miembro dato dentro de una definición de la clase, dado que la definición de la clase indica simplemente el *tipo* de cada miembro dato y no reserva realmente memoria. En su lugar, se desea inicializar los miembros dato cada vez que se crea una *instancia específica* de la clase. El sitio lógico para inicializar miembros dato, por consiguiente, está dentro del constructor de la clase. El constructor de la clase inicializa los miembros dato utilizando *expresiones de asignación*. Sin embargo, ciertos tipos de datos —específicamente, constantes y referencias— no pueden ser valores asignados. Para resolver este problema, C++ proporciona una característica de constructor especial conocido como *lista inicializadora de miembros* que permite *inicializar* (en lugar de asignar) a uno o más miembros dato.

Una lista inicializadora de miembros se sitúa inmediatamente después de la lista de parámetros en la definición del constructor; consta de un carácter dos puntos, seguido por uno o más *inicializadores de miembro*, separados por comas. Un inicializador de miembros consta del nombre de un miembro dato seguido por un valor inicial entre paréntesis.

Ejemplo

```
class C {
private:
   int T;
   const int CInt;
   int &Dint;
   // ...
public:
   C (int Param)-: T (Param), CInt(25), Dint (T)
   {
      // código del constructor
   }
   // ...
};
```

La definición siguiente crea un objeto

```
C CObjeto (0);
```

con los miembros dato `T` y `CInt` inicializadas a `0` y `25`, y el miembro dato `DInt` se inicializa de modo que se refiere a `T`.

Ejercicio 13.2

Diseñar y construir una clase contador que cuente cosas. (Cada vez que se produzca un suceso el contador se incrementa en 1.) El contador puede ser consultado para encontrar la cuenta actual.

```
// contador.cpp
// objeto representa una variable contador
#include <iostream>
using namespace std;

class Contador {
private:
   unsigned int cuenta;        // contar
public:
   Contador()          {cuenta = 0;}    // constructor
   void inc_cuenta()   {cuenta++;}      // cuenta
   int leer_cuenta()   {return cuenta;} // devuelve cuenta
};

void main()
{
   Contador c1, c2             // define e inicializa

   cout << "\nc1 = " << c1.leer_cuenta();
```

```
    cout << "\nc2 = " << c2.leer_cuenta();

    c1.inc_cuenta();            // incrementa c1
    c2.inc_cuenta();            // incrementa c2
    c2.inc_cuenta();            // incrementa c2

    cout << "\nc1 = " << c1.leer_cuenta(); // visualiza de nuevo
    cout << "\nc2 = " << c2.leer_cuenta();
}
```

La clase `Contador` tiene un elemento dato: `cuenta`, del tipo `unsigned int` (la cuenta siempre es positivo). Tiene tres funciones miembro: `Contador();` `inc_cuenta()`, que añade 1 a cuenta; y `leer_cuenta()`, que devuelve el valor actual de `cuenta`.

13.4. DESTRUCTORES

En una clase se puede definir también una función miembro especial conocida como *destructor*, que se llama automáticamente siempre que se destruye un objeto de la clase. El nombre del destructor es el mismo que el nombre de la clase, precedida con el carácter ~. Al igual que un constructor, un destructor se debe definir sin ningún tipo de retorno (ni incluso `void`); al contrario que un constructor, no puede aceptar parámetros. Sólo puede existir un *destructor*.

Ejemplo 13.8

```
class Demo
{
private:
   int datos;
public:
   Demo()  {datos = 0;}         // constructor
   ~Demo() {}                   // destructor
};
```

Regla

- Los destructores no tienen valor de retorno.
- Tampoco tienen argumentos.

El uso más frecuente de un destructor es liberar memoria que fue asignada por el constructor. Si un destructor no se declara explícitamente, C++ crea un vacío automáticamente.

Si un objeto tiene ámbito local, su destructor se llama cuando el control pasa fuera de su bloque de definición. Si un objeto tiene ámbito de archivo, el destructor se llama cuando termina el programa principal (`main`). Si un objeto se asignó dinámicamente (utilizando `new` y `delete`), el destructor se llama cuando se invoca el operador `delete`.

Ejemplo 13.9

```
// El destructor notifica cuándo se ha destruido un Punto
class Punto {
```

```
public:
   ~Punto()
   {
      cout << "Se ha llamado al destructor de Punto \n";
   }
   // ...
};
```

13.4.1. Clases compuestas

Una *clase compuesta* es aquella que contiene miembros dato que son asimismo objetos de clases. Antes de que el cuerpo de un constructor de una clase compuesta, se deben construir los miembros dato individuales en su orden de declaración.

La clase `Estudiante` contiene miembros dato de tipo `Expediente` y `Dirección`:

```
class Expediente {// ...}
class Direccion  {// ...};
class Estudiante {
public :
   Estudiante()
   {
      PonerId(0);
      PonerNotaMedia(0.0);
   }
   void PonerId(long);
   void PonerNotaMedia(float);
   private:
      long id;
      Expediente exp;
      Direccion dir;
      float NotMedia;
   };
```

Aunque `Estudiante` contiene `Expediente` y `Dirección`, el constructor de `Estudiante` no tiene acceso a los miembros privados o protegidos de `Expediente` o `Dirección`. Cuando un objeto `Estudiante` sale fuera de alcance, se llama a su destructor. El cuerpo de `~Estudiante` se ejecuta antes que los destructores de `Expediente` y `Dirección`. En otras palabras, el orden de las llamadas a destructores a clases compuestas es exactamente el opuesto al orden de llamadas de constructores.

13.5. SOBRECARGA DE FUNCIONES MIEMBRO

Al igual que sucede con las funciones no miembro de una clase, las funciones miembro de una clase se pueden sobrecargar. A excepción de los operadores sobrecargados —que tienen reglas especiales que se estudian en el Capítulo 22— una función miembro se puede sobrecargar pero sólo en su propia clase. Una función miembro de una clase no está relacionada y por ello no se puede sobrecarga, normalmente, con funciones no miembro o funciones declaradas en otras clases.

Las mismas reglas utilizadas para sobrecargar funciones ordinarias se aplican a las funciones miembro: dos miembros sobrecargados no puede tener el mismo número y tipo de parámetros. La sobrecarga permite utilizar un mismo nombre para una función y ejecutar la función definida más adecuada a los

parámetros parados durante la ejecución del programa. La ventaja fundamental de trabajar con funciones miembro sobrecargadas es la comodidad que aporta a la programación.

Para ilustrar la sobrecarga, veamos la clase `Producto` donde aparecen diferentes funciones miembro con el mismo nombre y diferentes tipos de parámetros

```
class Producto
{
public:
   int producto (int m, int n);                //funcion 1
   int producto (int m, int p, int q);         //funcion 2
   int producto (float m, float n);            //funcion 3
   int producto (float m, float n, float p);   //funcion 4
}
```

Registros para la sobrecarga

No pueden existir dos funciones en el mismo ámbito con igual signatura (lista de parámetros).

Los constructores suelen ser las funciones que se sobrecargan con mayor frecuencia, como ya se ha visto en ejemplos anteriores.

13.6. ERRORES DE PROGRAMACIÓN FRECUENTES

Las facilidades en el uso de la estructura *clase* aumenta las posibilidades de errores.

1. *Mal uso de palabras reservadas*. Es un error declarar una clase sin utilizar fielmente una de las palabras reservadas `class`, `struct` o `union` en la declaración.
2. La palabra reservada `class` no se necesita para crear objetos de clases. Por ejemplo, se puede escribir

```
class C {
   ...
};
C c1, c2            // definiciones de objetos
```

en lugar de

```
class C c1, c2      // no se necesita class
```

3. Si un miembro de una clase es privado (`private`), se puede acceder sólo por métodos de funciones amigas de la clase. Por ejemplo, este código no es correcto.

```
class C {
   int x;
   ...
public :
   C() {x = -9999;}
} ;

void f()
{
   C c;
```

```
   cout << c.x;          // ERROR
}
```

ya que `x` es privado en la clase `C` y la función `f` no es ni un método ni una amiga de `C`.

4. *Uso de constructores y destructores*.

   • No se puede especificar un tipo de retorno en un constructor o destructor, ni en su declaración ni en su definición. Si `C` es una clase, entonces no será legal:

   ```
   void C::C I(int n)         // ERROR
   {
      ...
   }
   ```

   • De igual modo, no se puede especificar ningún argumento en la definición o declaración de un constructor por omisión de la clase:

   ```
   C::C(void)         // ERROR
   {
      ...
   }
   ```

   • Tampoco se pueden especificar argumentos, incluso `void`, en la declaración o definición del destructor de una clase.
   • No se pueden tener dos constructores de la misma clase con los mismos tipos de argumentos.
   • Un constructor de una clase `C` no puede tener un argumento de un tipo `C`, pero sí puede tener un argumento de tipo referencia, `C&`. Este constructor es el constructor de copia.

5. *Inicializadores de miembros*.

   ```
   class C {
      // ERROR

      int x = 5;
   };
   class C {
      int x;
      ...
      public:
         C() {x = 5;}         // CORRECTO
   };
   ```

6. No se puede acceder a un miembro dato privado fuera de su clase, excepto a través de una función amiga (`friend`). Por ejemplo, el código

   ```
   class C {
      int x;
   };
   int main()
   {
      C c;
      c.x = 10;         // ERROR, x es privado en C
      ...
   }
   ```

   contiene un error, ya que `x` es privado en `C` y, por consiguiente, accesible sólo por los métodos o amigas de `C`.

7. No se puede acceder a un miembro dato protegido (`protected`) fuera de su jerarquía de clases, excepto a través de una función amiga. Por ejemplo,

```
class C {                    // clase base
protected:
   int x;
};

class D:public C {           // clase derivada
   ...
};

int main()
{
   C c1;
   c1.x = 10                 // ERROR, x es protegido
   ...
}
```

contiene un error, ya que `x` sólo es accesible por métodos de `C` y métodos y amigas de las clases derivadas de `C`, tales como `D`.

8. No se pueden sobrecargar ninguno de estos operadores:

```
.      .*      ::      ?: sizeof
```

9. No se puede utilizar `this` como parámetro, ya que es una palabra reservada. Tampoco se puede utilizar en una sentencia de asignación tal como

```
this = ...;      // this es constante; ERROR
```

ya que `this` es una constante.

10. Un constructor de una clase `C` no puede esperar un argumento de tipo `C`:

```
class C {
   ...
};
C::C(C c)                   // ERROR
{
   ...
}

C::C(C& c)                  // CORRECTO
{
   ...
}
```

11. Un miembro dato `static` no se puede definir dentro de una clase, aunque sí debe ser declarado dentro de la declaración de la clase.

```
class C {
   static int x = 7;        // ERROR
   ...
};

class D {
   static int x;
```

```
};

// definición con inicialización
int D::x = 7;
```

12. Los constructores o destructores no pueden declarar `static`.
13. *Olvido de puntos y coma en definición de clases*. Las llaves `{}` son frecuentes en código C++, y, normalmente, no se sitúa un punto y coma después de la llave de cierre. Sin embargo, la definición de `class` siempre termina en `};`. Un error típico es olvidar ese punto y coma.

```
class Producto
{
   public:
      // ...
   private:
      //...
} // olvido del punto y coma
```

14. *Olvido de inicialización de todos los campos de un constructor*. Todo constructor necesita asegurar que todos los campos (miembros) de datos se fijan a los valores apropiados.

```
class Empleado {
public:
   Empleado();
   Empleado(string n); // se olvida el parámetro salario
   //...
private:
   string nombre;
   float salario;
};
```

15. *Referencia a un atributo privado de una clase*. Los identificadores declarados como atributos o funciones privados de una clase no pueden ser referenciados desde el exterior de la clase. Cualquier intento de referencia producirán mensaje de error similar a

```
"undefined symbol"
```

16. *Fallo por no incluir un archivo de cabecera requerido*. Se generan numerosos mensajes de error por este fallo de programación ya que un archivo de cabecera contiene la definición de un número de los identificadores utilizados en su programa y el mensaje más frecuente de estos errores será:

```
undefined symbol
```

17. *Fallo al definir una función como miembro de una clase*. Este error se puede producir de varias formas:

    1. Fallo al prefijar la definición de la función por el nombre de su clase y el operador de resolución de ámbito (`::`).
    2. Fallo al escribir el nombre de la función correctamente.
    3. Omisión completa de la definición de la función de la clase.

    En cualquiera de los casos, el resultado es el mismo: un mensaje de error del compilador que indica que la función llamada no existe en la clase indicada.

```
"is not a member"
```

Ejemplo

Si se invoca a una función `imprimir` del objeto `c1` de la clase `contador` en la sentencia de C++:

```
c1.imprimir();
```

aparece en el programa `cliente`; se visualizará el mensaje

```
" 'imprimir' if not a member of 'contador'"
```

18. *Olvido de puntos y coma en prototipos y cabeceras de funciones*. La omisión de un punto y coma al final del prototipo de una función puede producir el mensaje de error

«`Statement missing`»

o bien

«`Declaration terminated incorrectly`».

RESUMEN

- Una **clase** es un tipo de dato definido por el usuario que sirve para representar objetos del mundo real.
- Un objeto de una clase tiene dos componentes: un conjunto de atributos y un conjunto de comportamientos (operaciones). Los atributos se llaman miembros dato y los comportamientos se llaman funciones miembro.

```
class circulo {
  double x_centro, y_centro;
  double superficie(void);
};
```

- Cuando se crea un nuevo tipo de clase, se deben realizar dos etapas fundamentales: determinar los atributos y el comportamiento de los objetos.
- Un objeto es una instancia de una clase.

```
circulo un_circulo, tipo_circulo[100];
```

- Una declaración de una clase se divide en tres secciones: *pública*, *privada* y *protegida*. La sección pública contiene declaraciones de los atributos y el comportamiento del objeto que son accesibles a los usuarios del objeto. Se recomienda la declaración de los constructores en la sección pública. La sección privada contiene las funciones miembro y los miembros dato que son ocultos o inaccesibles a los usuarios del objeto. Estas funciones miembro y atributos dato son accesibles sólo por la función miembro del objeto.
- El acceso a los miembros de una clase se puede declarar como *privado* (`private`, por defecto), *público* (`public`) o *protegido* (`protected`).

```
class circulo {
public:
   double superficie(void);
   void fijar_centro(double x,
      double y);
   void leer_radio(double radio)
      { r = radio; }
   double dar_radio(void);
private:
   double centro_x, centro_y;
   double r;
};
```

Los miembros dato `centro_x`, `centro_y` y `r` son ejemplos de ocultación de datos.

- Una función definida dentro de una clase, tal como `fijar_centro()`, se declara implícitamente en línea (`inline`). El operador de resolución de ámbito `::` se utiliza para definir una función miembro externa a la definición de la clase.

```
double circulo::dar_radio(void)
   { return r; }
```

Las funciones definidas así no son implícitamente `inline`.

- Existen dos métodos fundamentales de especificar un objeto

```
circulo c;           // un objeto
circulo *pt_objeto;  // un puntero a
                        un objeto
```

y dos métodos para especificar un miembro de una clase

```
radio = 10.0                    // Miembro
                                   de la
                                   clase
double circulo::pt_miembro;     // Puntero
                                   a un
                                   miembro
```

Se pueden utilizar dos operadores diferentes para acceder a miembros de la clase:

1. El operador de acceso a miembro (el operador punto).

   `c. radio = 10.0;`

2. El operador de acceso a miembros de la clase (el operador flecha).

   `pt_Objeto ⇐ radio = 10;`

3. Un operador puntero a miembro

   `pt_objeto ⇐ radio = 10;`

- Un **constructor** es una función miembro con el mismo nombre que su clase. Un constructor no puede devolver un tipo pero puede ser sobrecargado.

```
class complejo {
// ...
  complejo (double x, double y);
  complejo(const complejo &c);
```

Para una clase, `x`, un constructor con el prototipo, `x::x(const x&)`, se conoce como *constructor de copia*.

- Un **constructor** es una función miembro especial que se invoca cuando se crea un objeto. Se utiliza normalmente para inicializar los atributos de un objeto. Los argumentos por defecto hacen al constructor más flexible y útil.
- El proceso de crear un objeto se llama *instanciación* (creación de instancia).
- Una función miembro constante es una que no puede cambiar los atributos de sus objetos. Si se desea utilizar objetos de función clase `const`, se deben definir funciones miembro `const`.

```
inline double complejo::real(void)
   const
{
   return re;
}
```

- Un **destructor** es una función miembro especial que se llama automáticamente siempre que se destruye un objeto de la clase.

```
~complejo();
```

LECTURAS RECOMENDADAS

- Booch, G., *Object-Oriented Analysis and Design with Applications*, Reedwood City, CA (USA), Benjamin-Cummings, 1994.
- Rumbaugh, J.; Blaha, M; Premerlani, W.; Eddy, F., y Lorensen W., *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ (USA), Prentice-Hall, 1991.
- Wirfs-Brock, R.; Wilkerson, B., y Wiener, L., *Designing Object-Oriented Software*, Englewood Cliffs, NJ (USA), Prentice-Hall, 1990.
- Joyanes, Luis, *Programación Orientada a Objetos*, 2.ª edición, Madrid, McGraw-Hill, 1998.

EJERCICIOS

13.1. ¿Cuál es el error de la siguiente declaración de clase?

```
union float_bytes {
private :
   char mantisa[3], char exponente ;
public:
   float num;
   char exp(void);
   char mank(void);
};
```

13.2. ¿Qué está mal en la siguiente definición de la clase?

```
#include <string.h>

struct buffer {
   char datos[255];
   int cursor ;
   void iniciar(char *s);
   inline int Long(void);
   char *contenido(void);
};
void buffer::iniciar(char *s)
{
   strcpy(datos,s) ; cursor = 0;
}

char *buffer::contenido(void)
{
   if(Long()) return datos; else
      return 0;
}
inline int buffer:: Long(void)
   {return cursor;}
```

13.3. Consideremos una pila como un tipo abstracto de datos. Se trata de definir una clase que implementa una pila de 100 caracteres mediante un array. Las funciones miembro de la clase deben ser:

`meter, sacar, pilavacia y pilallena.`

13.4. Rescribir la clase pila que utilice una lista enlazada en lugar de un array. (Sugerencia: utilice otra clase para representar los modos de la lista.)

13.5. Examine la siguiente declaración de clase y vea si existen errores.

```
class punto {
   int x, y;
   void punto(int x1, int y1)
      {x = x1; y = y1;)
} ;
```

13.6. Dado el siguiente programa, ¿es legal la sentencia de main()?

```
class punto {
public:
   int x, int y;
   punto(int x1, int y1) {x = x1;
      y = y1;}
};

main()
{
   punto(25, 15);  //¿es legal esta
                     sentencia?
}
```

13.7. Suponiendo contestado el ejercicio anterior, ¿cuál será la salida del siguiente programa?

```
#include <stdio.h>

class punto{
public:
   int x, int y;
   punto(int x1, int y1)  {x = x1;
      y = y1}
   punto(int z)   {punto (z, z);}
};

main()
{
   punto p(25);
   printf("x = %d, y = %d\n", p.x,
       p.y);
}
```

13.8. Clasificar los siguientes constructores, cuyos prototipos son:

```
rect::rect(int a, int h);
nodo::nodo(void);
calculadora::calculadora(float *val
   i = 0.0);
cadena::cadena(cadena &c);
abc::abc(float f);
```

13.9. Dadas las siguientes declaraciones de clase y array, ¿por qué no se puede construir el array?

```
class punto {
public:
   int x, y;
   punto(int a, int b) {x = a;
      y = b;}
};

punto poligono[5];   // ¿es válida
                        esta sen-
                        tencia?
```

13.10. Se cambia el constructor del ejercicio anterior por un constructor por defecto. ¿Se puede construir, en este caso, el array?

```
class punto {
public:
   int x, y;
   punto(int a = 0, int b = 0)
      { x = a; y = b;}
};

punto poligono[5] ;   // ¿es válida
                         esta sen-
                         tencia?
```

13.11. ¿Cuál es el constructor que se llama en las siguientes declaraciones?

```
class prueba_total {
private:
   int num;
public:
   prueba_total(void)      {num = 0;}
   prueba_total(int n = 0) {num = n;}
   int valor(void)         {return num;}
};

prueba_total prueba;
```

13.12. Dado el siguiente programa C++, escribir un programa C equivalente.

```
#include <stdio.h>

clas operador {
public:
   float memoria;
   operador(void);
   operador(void);
   float sumar(float f);
};

operador::operador(void)
{
   printf("Activar maquina operador
      \n");
   memoria = 0.0
}

operador::˜operador(void)
{
   printf("Desactivar maquina ope-
      rador \n");
}

float operador::sumar(float f)
{
   memoria += f;
   return memoria;
}
main()
{
   operador o
   o.sumar(10.0);
   o.sumar(30.0);
   o.sumar(3.0);
   printf("la solucion es % f\n, τ
      .memoria);
}
```

13.13. Crear una clase llamada hora que tenga miembros datos separados de tipo int para horas, minutos y segundos. Un constructor inicializará este dato a 0, y otro lo inicializará a valores fijos. Una función miembro deberá visualizar la hora en formato 11:59:59. Otra función miembro sumará dos objetos de tipo hora pasados como argumentos. Una función principal main() crea dos objetos inicializados y uno que no está inicializado. Sumar los dos valores inicializados y dejar el resultado en el objeto no inicializado. Por último, visualizar el valor resultante.

13.14. Crear una clase llamada empleado que contenga como miembro dato el nombre y el número de empleado, y como funciones miembro `Leerdatos()` y `verdatos()` que lean los datos del teclado y los visualice en pantalla, respectivamente.

Escribir un programa que utilice la clase, creando un array de tipo empleado y luego llenándolo con datos correspondientes a 50 empleados. Una vez rellenado el arrays, visualizar los datos de todos los empleados.

13.15. Realizar un programa que calcule la distancia media correspondiente a 100 distancias entre ciudades dadas cada una de ellas en kilómetros y metros.

13.16. Se desea realizar una clase `vector3d` que permita manipular vectores de tres componentes (coordenadas `x`, `y`, `z`) de acuerdo a las siguientes normas:

- Sólo posee una función constructor y es en línea.
- Tiene una función miembro `igual` que permite saber si dos vectores tienen sus componentes o coordenadas iguales (la declaración de `igual` se realizará utilizando: *a*) transmisión por valor; *b*) transmisión por dirección; *c*) transmisión por referencia).

13.17. Incluir en la clase `vector3d` del ejercicio anterior una función miembro denominada `normamax` que permita obtener la norma mayor de dos vectores (Nota: La norma de un vector $v = x, y, z$ es x2+y2+z2 o bien `x*x+y*y+z*z.`)

13.18. Diseñar una clase `vector3d` que permita manipular vectores de 3 componentes (de tipo `float`) y que contenga una función constructor con valores por defecto (0) y las funciones miembros `suma` (suma de dos vectores), `productoescalar` (producto escalar de dos vectores-: $v1 = x1, y1, z1$; $v2 = x2, y2, z2$; $v1 * v2 = x1 * x2 + y1 * y2 + z1 * z2$).

13.19. Dada la siguiente clase punto:

```
class punto{
   int x, y;
public :
   punto(int abs = 0, int ord = 0)
   { x = abs; y = ord;}
};
```

Escribir una función independiente `visualizar`, amiga de la clase `punto`, que permite visualizar las coordenadas de un punto. Se proporcionará por separado un archivo fuente que contenga la nueva declaración (definición) de `punto` y un archivo fuente que contiene la definición de la función `visualizar`. Escribir un programa que cree un punto de la clase automática y un punto de clase dinámica y que visualice las coordenadas.

13.20. Cuál es la diferencia de significado entre la estructura

```
struct a {
   int i, j, k;
};
```

y la clase

```
class a {
   int i, j, k;
};
```

Explique la razón por la que la declaración de la clase no es útil. ¿Cómo se puede utilizar la palabra reservada `public` para cambiar la declaración de la clase en una declaración equivalente a `struct a`?

13.21. Realizar una clase `Complejo` que permita la gestión de números complejos (un número complejo = dos números reales `double`: una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

- Una función `establecer()` permite inicializar un objeto de tipo `Complejo` a partir de dos componentes `double`.
- Una función `imprimir()` realiza la visualización formateada de un `Complejo`.
- Dos funciones `agregar()` (sobrecargadas) permiten añadir, respectivamente, un `Complejo` a otro y añadir dos componentes `double` a un `Complejo`.

13.22. Crear una clase `lista` que realice las siguientes tareas:

- Una lista simple que contenga cero o más elementos de algún tipo específico.
- Crear una lista vacía.
- Añadir elementos a la lista.
- Determinar si la lista está vacía.
- Determinar si la lista está llena.
- Acceder a cada elemento de la lista y realizar alguna acción sobre ella.

PROBLEMAS

13.1. Implementar una clase `Random` (aleatoria) para generar números pseudoaleatorios.

13.2. Implementar una clase `Fecha` con miembros dato para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, un constructor de copia, funciones de acceso, una función `reiniciar (int d, int m, int a)` para reiniciar la fecha de un objeto existente, una función `adelantar(int d, int m, int a)`para avanzar una fecha existente (día, `d`, mes, `m`, y año `a`) y una función `imprimir()`. Utilizar una función de utilidad `normalizar()` que asegure que los miembros dato están en el rango correcto 1 ℑ año, 1 ℑ mes 12, día ℑ días (Mes), donde `dias(Mes)` es otra función que devuelve el número de días de cada mes.

13.3. Ampliar el programa anterior de modo que pueda aceptar años bisiestos. **Nota**: Un año es bisiesto si es divisible por 400, o si es divisible por 4 pero no por 100. Por ejemplo, el año 1992 y 2000 son años bisiestos y 1997 y 1900 no son bisiestos.

13.4. Definir una clase `Racional` que represente a números racionales (fracciones). Los miembros privados serán el numerador y el denominador de la fracción, y en la parte pública se debe disponer al menos de las siguientes funciones miembros: `asignar, convertir, invertir, imprimir`, que realizarán las funciones

de: asignar los valores de los parámetros a numerador y denominador respectivamente (por ejemplo, 22/7); convertir a decimal el número racional (por ejemplo, 3.14286); calcular el inverso de la fración (por ejemplo, 7/22) y por último, visualizar la fracción (por ejemplo, se ha de ver 22/7, 6/15, etc.).

13.5. Implementar una clase `Punto` que represente a puntos en tres dimensiones *(x, y, x)*. Incluir un constructor por defecto, un constructor de copia, una función `negar ()` que transforme el punto en su opuesto (negativo), una función `norma ()` que devuelva la distancia al punto desde el origen (0,0,0) y una función `visualizar ()`.

13.6. Implementar una clase `Hora`. Cada objeto de esta clase representa una hora específica de un día, almacenando las horas, minutos y segundos como enteros. Incluir un constructor, funciones de acceso, una función `avanzar ()` para avanzar (adelantar) la hora actual de un objeto existente, una función `poner_a_cero` para poner a cero la hora actual de un objeto existente y una función `visualizar()`.

13.7. Implementar una clase `Persona` que represente datos personales de una persona. Los miembros datos deben incluir el nombre de la persona, fecha de nacimiento y año de graduación en la universidad. Se debe incluir un constructor por defecto, un destructor, funciones de acceso y función de visualización.

13.8. Implementar una clase `Cadena`. Cada objeto de la clase representa una cadena de caracteres. Los miembros datos son la longitud de la cadena y la cadena de caracteres actual. Además, se deben añadir constructores, destructor, funciones de acceso y función de visualizar, así como incluir una función `caracter ()` que devuelva el carácter de la cadena representado por el parámetro `i` que representa el índice o lugar del carácter en la cadena.

13.9. Implementar una clase `Matriz` 2 × 2 que incluya un constructor por defecto, un constructor de copia, una función `inversa()` que devuelve el inverso de la matriz, una función `det()` que devuelva el determinante de la matriz, una función lógica (boolean) `EsCero` que devuelva `time` o `falso` de acuerdo a que el determinante sea cero y una función `visualizar()`.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 272).
2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

13.1. *¿Cuál es el error de la siguiente declaración de clase?*

```
union float_bytes
{
  private:
      char mantisa[3], char expo-
          nente ;
  public:
      float num;
      char exp(void);
      char mank(void);
};
```

13.2. *Examine la siguiente declaración de clase y vea si existen errores.*

```
class punto {
 int x, y;
 void punto(int x1, int y1)
    {x = x1; y = y1;}
};
```

13.3. *Dado el siguiente programa, ¿es legal la sentencia de main()?*

```
class punto {
public:
```

```
   int x;
   int y;
   punto(int x1, int y1) {x = x1;
      y = y1;}
};

main()
{
   punto(25, 15);  //¿es legal esta
                     sentencia?
}
```

13.4. *Dadas las siguientes declaraciones de clase y array. ¿Por qué no se puede construir el array?*

```
class punto {
public:
   int x, y;
   punto(int a, int b) {x = a-;
      y = b;}
};

punto poligono[5];
```

13.5. *Se cambia el constructor del ejercicio anterior por un constructor por defecto. ¿Se puede construir, en este caso, el array?*

13.6. *¿Cuál es el constructor que se llama en las siguientes declaraciones?*

```
class prueba_total
{
  private:
      int num;
  public:
      prueba_total(void)   {num = 0;}
      prueba_total(int n)  {num = n;}
      int valor(void)      {return num;}
};
prueba_total prueba
```

13.7. *¿Cuál es la diferencia de significado entre la estructura?*

```
struct a
 {
  int i, j, k;
 };
```

y la clase

```
class a
{
   int i, j, k;
};
```

Explique la razón por la cual la declaración de la clase no es útil. ¿Cómo se puede utilizar la palabra reservada `public` *para cambiar la declaración de la clase en una declaración equivalente a* `struct`*?*

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 273).

2. Sitio web del libro, *`www.mhe.es/joyanes`* (código fuente).

13.1. *Crear una clase llamada* `hora` *que tenga miembros datos separados de tipo* `int` *para horas, minutos y segundos. Un constructor inicializará este dato a 0, y otro lo inicializará a valores fijos. Una función miembro deberá visualizar la hora en formato 11:59:59. Otra función miembro sumará dos objetos de tipo* hora *pasados como argumentos. Una función principal* `main()` *crea dos objetos inicializados y uno que no está inicializado. Sumar los dos valores inicializados y dejar el resultado en el objeto no inicializado. Por último, visualizar el valor resultante.*

13.2. *Un número complejo tiene dos partes: una parte real y una parte imaginaria; por ejemplo, en (4.5+3.0i), 4.5 es la parte real y 3.0 es la parte imaginaria. Realizar una clase* `Complejo` *que permita la gestión de números complejos (un número complejo = dos números reales). Las operaciones a implementar son las siguientes:*

- *Una función* `leerComplejo()` *permite leer un objeto de tipo* `Complejo`.
- *Una función* `escribirComplejo()` *realiza la visualización formateada de un* `Complejo`.

13.3. *Suponga que* `a = (A, Bi)` *y* `c = (C, Di)`. *Se desea añadir a la clase* `complejo` *las operaciones:*

- *Suma:* `a + c = (A + C, (B + D)i).`
- *Resta:* `a – c = (A – C, (B – D)i).`
- *Multiplicación:* `a * c=(A * C – B * D, (A * D + B * C)i).`
- *Multiplicación:* `x * c = (x * C, x * Di)`, *donde* `x` *es real*.
- *Conjugado:* `~a = (A. –Bi)`.

13.4. *Escribir una clase* `Conjunto` *que gestione un conjunto de enteros (*`int`*) con ayuda de una tabla de tamaño fino (un* conjunto *contiene una lista ordenada de elementos y se caracteriza por el hecho de que cada elemento es único: no se debe encontrar dos veces el mismo valor en la tabla). Las operaciones a implementar son las siguientes:*

- *La función* `vacía()` *vacía el conjunto.*
- *La función* `agregar()` *añade un entero al conjunto.*
- *La función* `eliminar()` *retira un entero del conjunto.*
- *La función* `copiar()` *recopila un conjunto en otro.*
- *La función* `es_miembro()` *reenvía un valor booleano (valor lógico que indica si el conjunto contiene un entero dado).*
- *La función* `es_igual()` *reenvía un valor booleano que indica si un conjunto es igual a otro.*
- *La función* `imprimir()` *realiza la visualización «formateada» del conjunto.*

13.5. *Añadir a la clase* `Conjunto` *del ejercicio anterior las funciones* `es vacio`, *y* `cardinal` *de un conjunto, así como, la unión, intersección, diferencia y diferencia simétrica de conjuntos.*

CAPÍTULO 14

Clases derivadas: herencia y polimorfismo

Contenido

INTRODUCCIÓN

En este capítulo se introduce el concepto de *herencia* y se muestra cómo crear *clases derivadas*. La herencia hace posible crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. El soporte de la herencia es una de las propiedades que diferencia los lenguajes *orientados a objetos* de los lenguajes *basados en objetos* y *lenguajes estructurados*.

La *herencia* es la propiedad que permite definir nuevas clases usando como base clases ya existentes. La nueva clase (*clase derivada*) hereda los atributos y comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, de bajo coste, adaptable y reutilizable.

CONCEPTOS CLAVE

- Clase abstracta.
- Clase base.
- Clase derivada.
- Constructor.
- Declaración de acceso.
- Destructor.
- Especificadores de acceso.
- Función virtual.
- Herencia.
- Herencia múltiple.
- Herencia pública y privada.
- Herencia simple.
- Ligadura dinámica.
- Polimorfismo.
- Relación *es-un*.
- Relación *tiene-un*.

14.1. CLASES DERIVADAS

La *herencia* o relación **es-un** es la relación que existe entre dos clases, en la que una clase denominada *derivada* se crea a partir de otra ya existente, denominada *clase base*. Este concepto nace de la necesidad de construir una nueva clase y existe una clase que representa un concepto más general; en este caso la nueva clase puede *heredar* de la clase ya existente. Así por ejemplo, si existe una clase `Figura` y se desea crear una clase `Triángulo`, esta clase `Triángulo` puede derivarse de `Figura` ya que tendrá en común con ella un estado y un comportamiento, aunque luego tendrá sus características propias. `Triángulo` *es-un* tipo de `Figura`. Otro ejemplo puede ser *`Programador`*, que *es-un* tipo de *`Empleado`*.

Evidentemente, la clase base y la clase derivada tienen código y datos comunes, de modo que si se crea la clase derivada de modo independiente, se duplicaría mucho de lo que ya se ha escrito para la clase base. C++ soporta el mecanismo de *derivación* que permite crear clases derivadas, de modo que la nueva clase *hereda* todos los miembros datos y las funciones miembro que pertenecen a la clase ya existente.

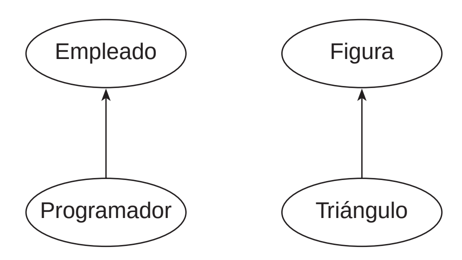


Figura 14.1. Clases derivadas.

La declaración de derivación de clases debe incluir el nombre de la clase base de la que se deriva y el especificador de acceso que indica el tipo de herencia (*pública*, *privada* y *protegida*). La primera línea de cada declaración debe incluir el formato siguiente:

```
class nombre_clase : tipo_herencia nombre_clase_base
```

Regla

En general, se debe incluir la palabra reservada **`public`** en la primera línea de la declaración de la clase derivada, y representa herencia pública. Esta palabra reservada consigue que todos los miembros que son públicos en la clase base permanezcan públicos en la clase derivada.

Ejemplo 14.1

Declaración de las clases `Programador` y `Triangulo`.

```
1. class Programador : public Empleado {
   public:
      // miembros públicos
   private:
      // miembros privados
   };
```

```
2. class Triangulo : public Figura
   {
      public:
         // sección pública
      ...
      private:
      // sección privada
      ...
   };
```

Una vez que se ha creado una clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.

```
              clase derivada              clase base

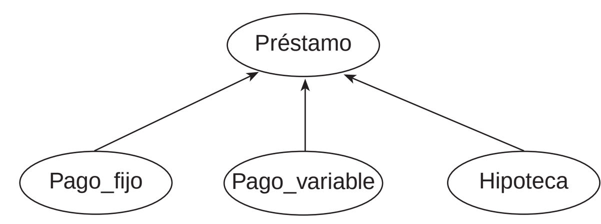
class Director : public  Empleado
{
   public:
      nuevas funciones miembro
   private:
      nuevos miembros dato
};
```

En la definición de la clase `Director` sólo se especifican los miembros nuevos (funciones y datos). Todas las funciones miembro y los miembros dato de la clase `Empleado` son heredados automáticamente por la clase `Director`. Por ejemplo, la función `calcular_salario` de `Empleado` se aplica automáticamente a los directores:

```
Director d;
d.calcular_salario(325000);
```

Ejemplo 14.2

Considérese una clase `Prestamo` *y tres clases derivadas de ella:* `Pago_fijo`, `Pago_variable` *e* `Hipoteca`.




```
class Prestamo }
protected:
   float capital;
   float tasa_interes;
public:
   Préstamo(float, float);
   virtual int CrearTablaPagos(float[MAX_TERM][NUM_COLUMNAS])=0;
};
```

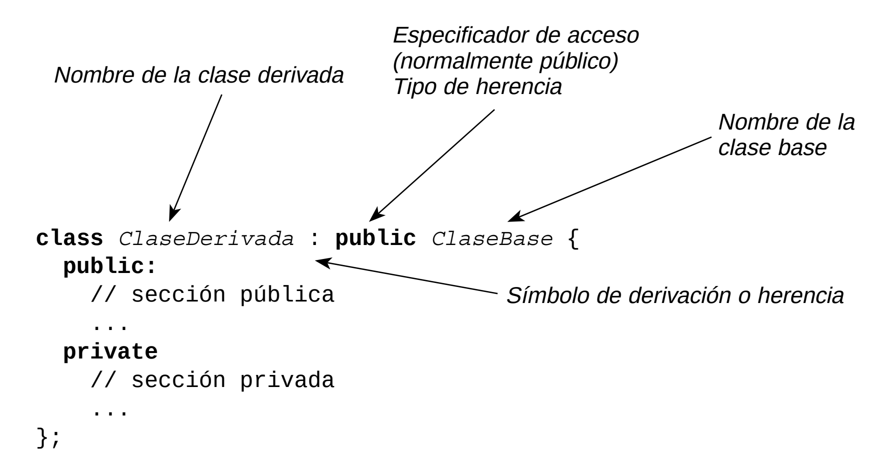
Las variables `capital` y `tasa_interes` no se repiten en la clase derivada

```
class Pago_fijo : public Préstamo {
private:
   float pago;      // cantidad mensual a pagar por cliente
public:
      Pago_Fijo (float, float, float);
      int CrearTablaPagos(float, [MAX_TERM][NUM_COLUMNAS]);
};

class Hipoteca : public Prestamo {
private:
   int num_recibos;
   int recibos_por_anyo;
   float pago;
public:
   Hipoteca(int, int, float, float, float);
   int CrearTablaPagos(float [MAX_TERN][NUM_COLUMNAS]);
};
```

14.1.1. Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es:



Especificador de acceso `public`, significa que los miembros públicos de la clase base son miembros públicos de la clase derivada.

Herencia pública, es aquélla en que el especificador de acceso es `public` (*público*).

Herencia privada, es aquélla en que el especificador de acceso es `private` (*privado*).

Herencia protegida, es aquélla en que el especificador de acceso es `protected` (*protegido*).

El especificador de acceso que declara el tipo de herencia es opcional (`public`, `private` o `protected`); si se omite el especificador de acceso, se considera por defecto `private`. La *clase base* (`ClaseBase`) es el nombre de la clase de la que se deriva la nueva clase. La *lista de miembros* consta de datos y funciones miembro:

```
class nombre_clase : especificador_acceso opcional ClaseBase {
   lista_de_miembros;
};
```

Ejercicio 14.1

Representar la jerarquía de clases de publicaciones que se distribuyen en una librería: revistas, libros, etc.

Todas las publicaciones tienen en común una editorial y una fecha de publicación. Las revistas tienen una determinada periodicidad lo que implica el número de ejemplares que se publican al año, y por ejemplo, el número de ejemplares que se ponen en circulación controlados oficialmente (por ejemplo, en España la OJD). Los libros, por el contrario tienen un código de ISBN y el nombre del autor



Las clases en C++ se especifican así:

```
class Publicacion {
public:
   void NombrarEditor(const char *s);
   void PonerFecha(unsigned long fe);

private:
   string editor;
   unsigned long fecha;
};

class Revista : public Publicacion {
public:
   void NumerosPorAnyo(unsigned n);
   void FijarCirculacion(unsigned long n);

private:
   unsigned numerosPorAnyo;
   unsigned long circulacion;
};

class Libro : public Publicacion {
public:
   void PonerISBN(const char *s);
   void PonerAutor (const char *s);

private:
   Dstring ISBN;
   Dstring autor;
};
```

Así, en el caso de un objeto `Libro`, éste contiene miembros dato y funciones heredadas del objeto `Publicación`, así como `ISBN` y nombre del `autor`. En consecuencia serán posibles las siguientes operaciones:

```
Libro L;
L.NombrarEditor("McGraw-Hill");
L.PonerFecha(990606);
L.PonerISBN("84-481-2015-9");
L.PonerAutor("Mackoy, José Luis");
```

Por el contrario, las siguientes operaciones sólo se pueden ejecutar sobre objetos `Revista`:

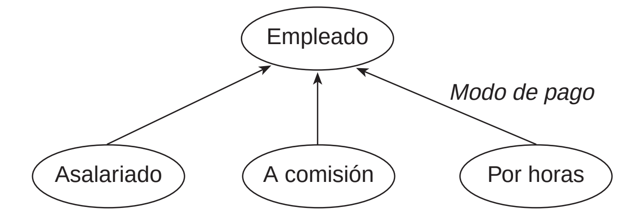
```
Revista R;
L.NumerosPorAnyo(12);
L.FijarCirculacion(300000L);
```

Si no existe la posibilidad de utilizar la herencia, sería necesario hacer una copia del código fuente de una clase, darle un nuevo nombre y añadirle nuevas operaciones y/o miembros dato. Esta situación provocaría una difícil situación de mantenimiento, ya que siempre que se hacen cambios en la clase original, los correspondientes cambios tendrán que hacerse también en cualquier «clase copiada».

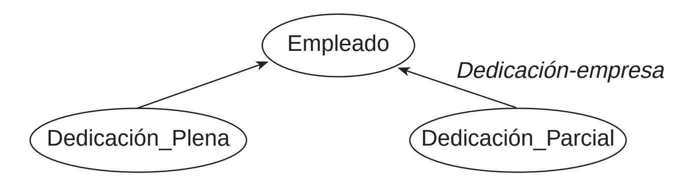
14.1.2. Consideraciones de diseño

A veces, es difícil decidir cuál es la relación de herencia más óptima entre clases en el diseño de un programa. Consideremos, por ejemplo, el caso de los empleados o trabajadores de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección (se suele llamar *discriminador*) y pueden ser: modo de pago (sueldo fijo, por horas, a comisión); dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

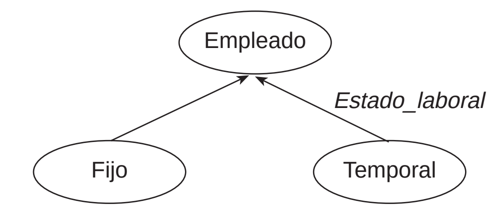
Una vista de los empleados basada en el modo de pago puede dividir a los empleados con salario mensual fijo; empleados con pago por horas de trabajo y empleados a comisión por las ventas realizadas.



Una vista de los empleados basada en el estado de dedicación a la empresa: dedicación plena o dedicación parcial.



Una vista de empleados basada en el estado laboral del empleado con la empresa: fija o temporal.



Una dificultad a la que suele enfrentarse el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Un empleado con dedicación plena puede ser remunerado con un salario mensual. Un empleado con dedicación parcial puede ser remunerado mediante comisiones y un empleado fijo puede ser remunerado por horas. Una pregunta usual es, ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones? ¿Esta relación ha de ser el fundamento del diseño de clases? Evidentemente, la respuesta adecuada sólo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

14.2. TIPOS DE HERENCIA

En una clase existen secciones públicas, privadas y protegidas. Los elementos públicos son accesibles a todas las funciones; los elementos privados son accesibles sólo a los miembros de la clase en que están definidos y los elementos protegidos pueden ser accedidos por clases derivadas debido a la propiedad de la herencia. En correspondencia con lo anterior, existen tres tipos de herencia: *pública, privada* y *protegida*. Normalmente, el tipo de herencia más utilizada es la herencia pública.

Con independencia del tipo de herencia, una clase derivada no puede acceder a variables y funciones privadas de su clase base. Para ocultar los detalles de la clase base y de clases y funciones externas a la jerarquía de clases, una clase base utiliza normalmente elementos protegidos en lugar de elementos privados. Suponiendo herencia pública, los elementos protegidos son accesibles a las funciones miembro de todas las clases derivadas.

Tabla 14.1. Acceso a variables y funciones según tipo de herencia.

Tipo de herencia	Tipo de elemento	¿Accesible a clase derivada?
`Public`	`public`	*sí*
	`protected`	*sí*
	`private`	*no*
`Private`	`public`	*sí*
	`protected`	*sí*
	`private`	*no*

Norma

Por defecto, la herencia es privada. Si accidentalmente se olvida la palabra reservada `public`, los elementos de la clase base serán inaccesibles. El tipo de herencia es, por consiguiente, una de las primeras cosas que se debe verificar, si un compilador devuelve un mensaje de error que indique que las variables o funciones son inaccesibles.

14.2.1. Herencia pública

En general, *herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y protegidos de su clase base. Los elementos públicos se heredan como elementos públicos; los elementos protegidos permanecen protegidos.

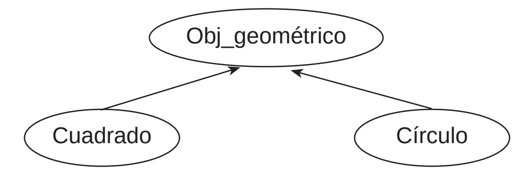
La herencia pública se representa con el especificador `public` en la derivación de clases.

Formato

```
class ClaseDerivada : public ClaseBase {
public:
   // sección pública
private:
   // sección privada
};
```

Ejercicio 14.2

Considérese la jerarquía `Obj_geométrico`, `Cuadrado` *y* `Círculo`.



La clase `Obj_geom` de objetos geométricos se declara como sigue:

```
#include <iostream>
using namespace std;

class obj_geom {
public:
   obj_geom(float x=0, float y=0) : xC(x), yC(y) {}
   void imprimircentro() const
   {
      cout << xC << " " << yC << endl;
   }
protected:
   float xC, yC;
};
```

Un círculo se caracteriza por su centro y su radio. Un cuadrado se puede representar también por su centro y uno de sus cuatro vértices. Declaremos las dos figuras geométricas como clases derivadas.

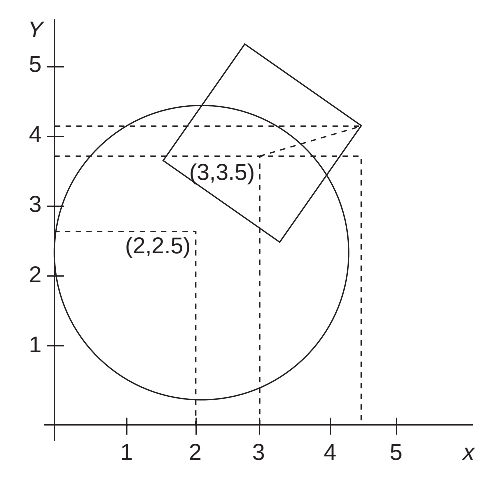


Figura 14.2. Círculo (*centro: 2, 2.5*), cuadrado (*centro: 3, 3.5*).

```
const float PI = 3.14159265;

class circulo : public obj_geom {
public:
   circulo(float x_C, float y_C, float r) : obj_geom (x_C, y_C)
   {
      radio = r;
   }
   float area() const {return PI * radio * radio; }
private:
   float radio;
};

class cuadrado : public obj_geom {
public :
   cuadrado(float x_C, float y_C, float x, float y)
             : obj_geom(x_C, y_C)
   {
      x1 = x;
      y1 = y;
   }
   float area() const
   {
      float a, b
      a = x1 - xC;
      b = y1 - yC;
      return 2 * (a * a + b * b);
}
private:
   float x1, y1;
};
```

Todos los miembros públicos de la clase base `obj_geom` se consideran también como miembros públicos de la clase derivada `cuadrado`. La clase `cuadrado` deriva públicamente de `obj_geom`. Se puede escribir

```
cuadrado C(3, 3.5, 4.37, 3.85);
C.imprimircentro();
```

Aunque `imprimircentro` no sucede directamente en la declaración de la clase `cuadrado`, es, no obstante, una de sus funciones miembro públicas ya que es un miembro público de la clase `obj_geom` de la que se deriva públicamente `cuadrado`. Otro punto observado es el uso de `xC` e `yC` en la función miembro `area` de la clase cuadrado. Éstos son miembros protegidos de la clase base `obj_geom`, por lo que tienen acceso a ellos desde la clase derivada.

Una función `main` que utiliza las clases `cuadrado` y `círculo` y la salida que se produce tras su ejecución:

```
int main()
{
   círculo C(2, 2.5, 2);
   cuadrado Cuad(3, 3.5, 4.37, 3.85);
   cout << "centro del circulo : "; C.imprimircentro();
```

```
   cout << "centro del cuadrado : " << Cuad.imprimircentro();
   cout << "Area del circulo : " << C.area() << endl;
   cout << "Area del cuadrado : " << Cuad.area() << endl;
   return 0;
}
```

```
Centro del circulo : 2 2.5
Centro del cuadrado : 3 3.5
Area del circulo : 12.5664
Area del cuadrado :3.9988
```

Regla

Con herencia pública, los miembros, de la clase derivada, heredados de la clase base tienen la misma protección que en la clase base. La herencia pública se utiliza casi siempre en la práctica ya que modela directamente la relación ***es-un***.

14.2.2. Herencia privada

La herencia privada significa que un usuario de la clase derivada no tiene acceso a ninguno de sus elementos de la clase base. El formato es:

```
class ClaseDerivada : private ClaseBase {
public:
   // sección pública
protected:
   // sección protegida
private:
   // sección privada
};
```

Con herencia privada, los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada. En efecto, los usuarios de la clase derivada no tienen acceso a las facilidades proporcionadas por la clase base. Los miembros privados de la clase base son inaccesibles a las funciones miembro de la clase derivada.

La herencia privada se utiliza con menos frecuencia que la herencia pública. Este tipo de herencia oculta la clase base del usuario y así es posible cambiar la implementación de la clase base o eliminarla toda junta sin requerir ningún cambio al usuario de la interfaz. Cuando un especificador de acceso no está presente en la declaración de una clase derivada, se utiliza herencia privada.

14.2.3. Herencia protegida

Con herencia protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada y los miembros privados de la clase base se vuelven inaccesibles. La herencia protegida es apropiada cuando las facilidades o aptitudes de la clase base son útiles en la implementación de la clase derivada, pero no son parte de la interfaz que el usuario de la clase ve. La herencia protegida es todavía menos frecuente que la herencia privada.

La Tabla 14.2 resume los efectos de los tres tipos de herencia en la accesibilidad de los miembros de la clase derivada. La entrada *inaccesible* indica que la clase derivada no tiene acceso al miembro de la clase base.

Tabla 14.2. Tipos de herencia y accesos que permiten.

Tipo de herencia	Acceso a miembro clase base	Acceso a miembro a clase derivada
`public`	`public` `protected` `private`	`public` `protected` *inaccesible*
`protected`	`public` `protected` `private`	`protected` `protected` *inaccesible*
`private`	`public` `protected` `private`	`private` `private` *inaccesible*

Ejemplo 14.3

Declarar una clase base (`Base`) y tres clases derivadas de ella, `D1`,`D2` y `D3`.

```
class Base {
public:
   int i1;
protected:
   int i2;
private:
   int i3;
};
class D1 : private Base {
   void f();
};
class D2 : protected Base {
   void g();
};
class D3 : public Base {
   void h();
};
```

Ninguna de las subclases tienen acceso al miembro `i3` de la clase `Base`. Las tres clases pueden acceder a los miembros `i1` e `i2`. En la definición de la función miembro `f()` se tiene:

```
void D1::f() {
   i1 = 0;          // Correcto
   i2 = 0;          // Correcto
   i3 = 0;          // Error
};
```

El acceso a `i1`, `i2` e `i3` desde el exterior de las tres clases se muestra en el siguiente programa:

```
void main()
{
   Base b;
   b.i1 = 0;       // Correcto
```

```
   b.i2 = 0;       // ERROR
   b.i3 = 0;       // ERROR
   D1 d1-;
   d1.i1 = 0;      // ERROR
   d1.i2 = 0;      // ERROR
   d1.i3 = 0;      // ERROR
   D2 d2;
   d2.i1 = 0;      // ERROR
   d2.i2 = 0;      // ERROR
   d2.i3 = 0;      // ERROR
   D3 d3;
   d3.i1 = 0;      // Correcto
   d3.i2 = 0;      // ERROR
   d3.i3 = 0;      // ERROR
};
```

14.2.4. Operador de resolución de ámbito

Si se utiliza herencia privada o protegida, existe un método de hacer a los miembros de la clase base accesibles en una clase derivada: utilizar una *declaración de acceso*. Esto se consigue nombrando uno de los miembros de la clase base en un lugar apropiado de la clase derivada.

```
class D4 : protected Base {
public
   Base::i1;           // declaración de acceso
};
```

Al declarar `i1` en la sección pública de `D4` también hace a `i1` público en `D4`. Se puede, entonces, escribir

```
D4 d4;
d4.i1 = 0;            // CORRECTO
```

De modo similar se puede hacer a `i2` protegido en `D5` nombrando `i2` en la sección protegida de `D5`.

```
class D5 : private Base {
protected:
   Base::i2;     // declaración de acceso
};
```

14.2.5. Constructores-inicializadores en herencia

Una clase derivada es una especialización de una clase base. En consecuencia, el constructor de la clase base debe ser llamado para crear un objeto de la clase base antes de que el constructor de la clase derivada realice su tarea. Haciendo un símil sucede lo mismo que con objetos de las clases derivadas; el objeto de la clase base debe existir antes de convertirse en un objeto de la clase derivada.

Regla

1. Los constructores de las clases base se invocan antes del constructor de la clase derivada; los constructores de la clase base se invocan en la secuencia en que están especificados.

2. Si una clase base es, a su vez, una clase derivada, sus constructores se invocan también en secuencia: constructor base, constructor derivada.
3. Los constructores no se heredan, aunque los constructores por defecto y de copia, se generan si se requiere.

Ejemplo 14.4

```
class B1 {
public:
   B1() { cout << "C-B1" << endl; }
};

class B2 {
public:
   B2() { cout << "C-B2" << endl; }
};

class D : public B1, B2 {
public:
   D() { cout << "C-D" << endl; }
};

D d1;
```

Este ejemplo visualiza: `C-B1, C-B2, C-D          //un resultado por línea`

Ejemplo 14.5

```
class B1 {
public:
   B1() { cout << "C-B1" << endl; }
};

class B2 {
public:
   B2() { cout << "C-B2" << endl; }
};

class D1 : public B1 {
public :
   D1() { cout << "C-D1" << endl; }
};

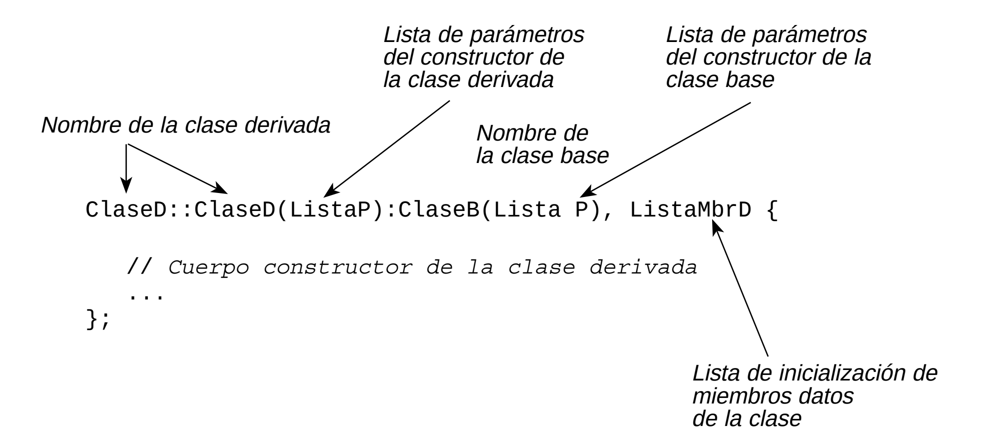
class D2 : public D1, public B2 {
public:
   D2() { cout << "C-D2" << endl; }
};

D2 d2;
```

El ejemplo visualiza: `C-B1, C-D1, C-B2 y C-D2 //un resultado por línea`

14.2.6. Sintaxis del constructor

La sintaxis de un constructor de una clase derivada es:



Obsérvese, que la primera línea incluye una llamada al constructor de la clase base. El constructor de la clase base se llama antes de que se ejecute el cuerpo del constructor de la clase derivada. Esta secuencia tiene sentido ya que el objeto base constituye el fundamento del objeto derivado (se necesita el objeto base antes de convertirse en objeto derivado). El constructor de una clase derivada tiene que realizar dos tareas:

- Inicializar el objeto base.
- Inicializar todos los miembros dato.

La clase derivada tiene un *constructor-inicializador*, que llama a uno o más constructores de la clase base. El inicializador aparece inmediatamente después de los parámetros del constructor de la clase derivada y está precedido por dos puntos (`:`).

Ejemplo 14.6

La clase `Punto3D` *es una clase derivada de la clase* `Punto`.

Formato general: `Punto3D::Punto3D(listaP):`*`inicializador-constructor`*

En la implementación del constructor de `Punto3D` se pasan los parámetros `xv` e `yv` al constructor de `Punto`.

```
class Punto {
public:
   Punto(int xv, int yv);
   // ...
};

class Punto3D: public Punto {
public:
   Punto3D(int xv, int yv, int zv);
   void FijarZ();
```

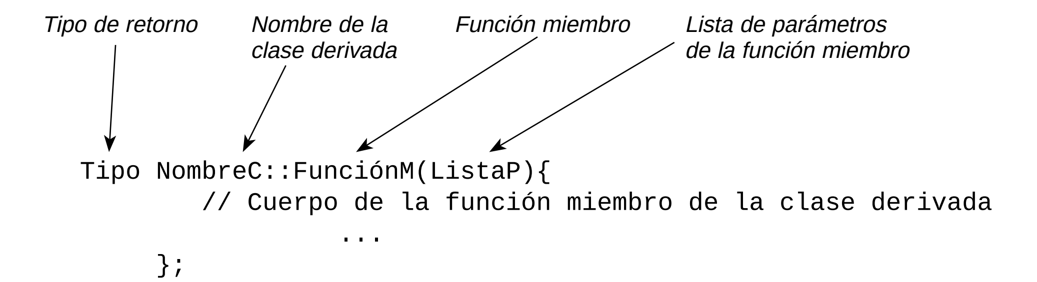
```
private:
   int z;
};

Punto3D ::Punto3D(int xv, int yv, int zv): Punto(xv, yv){
   FijarZ(zv);
}
```

Se ha implementado `Punto3D` fuera de la definición de la clase para mostrar cómo el constructor-inicializador no aparece en el prototipo de la función. Si `Punto3D` se implementara en el interior de la definición de la clase, el inicializador aparecería allí.

14.2.7. Sintaxis de la implementación de una función miembro

La sintaxis para la definición de la implementación de las funciones miembro de una clase derivada es idéntica a la sintaxis de la definición de la implementación de una clase base.



14.3. DESTRUCTORES

Afortunadamente, los destructores son mucho más fáciles de tratar que los constructores. Los destructores no se heredan, aunque se genera un destructor por defecto si se requiere. Un destructor normalmente sólo se utiliza cuando un constructor correspondiente ha asignado espacio de memoria que debe ser liberado. Los destructores se manejan como los constructores excepto que todo se hace en orden inverso. Esto significa que el destructor de la última clase derivada se ejecuta primero.

Ejemplo 14.7

```
class C1 {
public:
   C1(int n);
   ˜C1();
private:
   int *pi, l;
};

C1::C1(int n) :l(n)
{
   cout << l << " enteros se asignan " << endl;
   pi = new int[l];
}
```

```
C1::˜C1()
{
   cout << l << " enteros son liberados " << endl;
   delete[] pi;
}

class C2 : public C1 {
public :
   C2(int n);
   ˜C2();
private :
   char *pc;
   int l;
} ;
C2 ::C2(int n) :C1(n), l(n)
{
   cout << l << " caracteres son asignados " << endl;
   pc = new char[l];
}
C2::˜C2()
{
   cout << l << " caracteres son liberados " << endl;
   delete[] pc;
}

void main()
{
   C2 a(50), b(100);
}
```

Cuando de ejecuta el programa, se imprimirá:

```
50  enteros se asignan
50  caracteres se asignan
100 enteros se asignan
100 caracteres se asignan
100 caracteres son liberados
100 enteros son liberados
50  caracteres son liberados
50  enteros son liberados
```

14.4. HERENCIA MÚLTIPLE

Herencia múltiple es un tipo de herencia en la que una clase hereda el estado (estructura) y el comportamiento de más de una clase base. En otras palabras, hay herencia múltiple cuando una clase hereda de más de una clase; es decir, existen múltiples clases base (*ascendientes* o *padres*) para la clase derivada (*descendiente* o *hija*).

La herencia múltiple entraña un concepto más complicado que la herencia simple, no sólo con respecto a la sintaxis sino también al diseño e implementación del compilador. La herencia múltiple también aumenta las operaciones auxiliares y complementarias y produce ambigüedades potenciales. Además, el

diseño con clases derivadas por derivación múltiple tiende a producir más clases que el diseño con herencia simple. Sin embargo, y pese a los inconvenientes y ser un tema controvertido, la herencia múltiple puede simplificar los programas y proporcionar soluciones para resolver problemas difíciles. En la Figura 14.3 se muestran diferentes ejemplos de herencia múltiple.

Regla

En herencia simple, una clase derivada hereda exactamente de una clase base (tiene sólo un padre). Herencia múltiple implica múltiples clases bases (tiene varios padres una clase derivada).

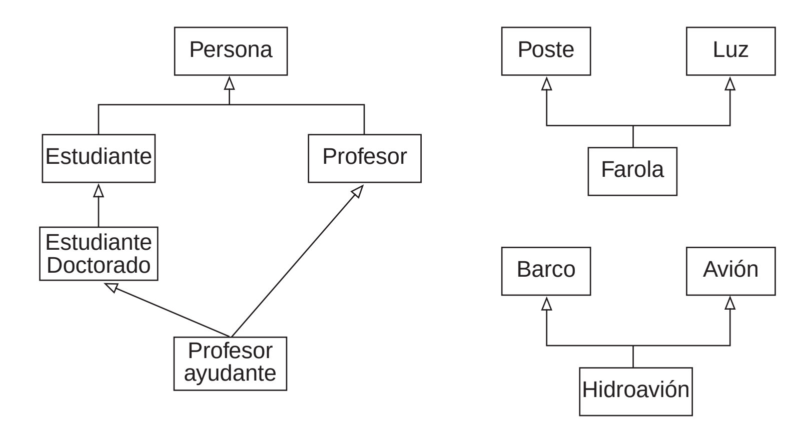


Figura 14.3. Ejemplos de herencia múltiple.

En herencia simple, el escenario es bastante sencillo, en términos de concepto y de implementación. En herencia múltiple los escenarios varían ya que las clases bases pueden proceder de diferentes sistemas y se requiere a la hora de la implementación un compilador de un lenguaje que soporte dicho tipo de herencia (C++ o Eiffel). ¿Por qué utilizar herencia múltiple? Pensamos que la herencia múltiple añade fortaleza a los programas y si se tiene precaución en el momento del análisis y posterior diseño, ayuda bastante a la resolución de muchos problemas que tienen naturaleza de herencia múltiple.

Por otra parte, la herencia múltiple siempre se puede eliminar y convertirla en herencia simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real. La sintaxis de la herencia múltiple es:

```
class Cderivada :   [virtual][tipo_acceso] Base1,
                    [virtual][tipo_acceso] Base2,
                    [virtual][tipo_acceso] Basen {
public:
   // sección pública
private:
   // sección privada
...
};
```

CDerivada	Nombre de la clase derivada
tipo_acceso	`public`, `private` o `protected`, con las mismas reglas que la herencia simple

`Base1, Base2,...`	Clases base con nombres diferentes
`virtual..`	La palabra reservada `virtual` es opcional y especifica una clase base compatible.

Funciones o datos miembro que tengan el mismo nombre en `Base1, Base2`, `Basen,` serán motivo de ambigüedad.

Ejemplos

```
class A : public B, public c {...};
class D : public E, private F, public G {...};
class X : Y, Z {...};
class M : virtual public N, virtual public P {...};
```

La palabra reservada `public` ya se ha comentado anteriormente, define la relación «***es-un***» y crea un subtipo para herencia simple. Así, en los ejemplos anteriores, la clase `A` «***es-un***» tipo de `B` y «***es-un***» tipo de `C`. La clase `D` se deriva públicamente de `E` y `G` y privadamente de `F`. Esta derivación hace a `D` un subtipo de `E` y `G` pero no un subtipo de `F`. La clase `X` se deriva privadamente de `Y` y `Z`, por defecto. Las clases `N` y `P` son clases base virtuales de `M` (se examinará más tarde el concepto). *El tipo de acceso sólo se aplica a una clase base*.

Ejemplo

```
class Derivada : public Base1, Base2 {...};
```

`Derivada` especifica derivación pública de `Base1` y derivación privada (por defecto u omisión) de `Base2.`

Regla

Asegúrese de especificar un tipo de acceso en todas las clases base para evitar el acceso privado por omisión. Utilice explícitamente `private` cuando lo necesite para manejar la legibilidad.

```
class Derivada : public Base1, private Base2 {...}
```

Ejemplo

```
class estudiante {
   ...
};

class trabajador {
   ...
};

class estudiante_trabajador : public estudiante, public trabajador {
   ...
};
```

14.4.1. Características de la herencia múltiple

La herencia múltiple plantea diferentes problemas tales como la *ambigüedad* por el uso de nombres idénticos en diferentes clases base, y la *dominación* o *preponderancia* de funciones o datos.

Ambigüedades

Al contrario que la herencia simple, la herencia múltiple tiene el problema potencial de las ambigüedades.

Ejemplo

```
class Ventana {
private:
   ...
public :
   void dimensionar();      // dimensiona una ventana
   ...
};

class Fuente {
private:
   ...
public:
   void dimensionar();      // dimensiona un tipo fuente
   ...
};
```

Una clase `Ventana` tiene una función `dimensionar()` que cambia el tamaño de la ventana; de modo similar, una clase `Fuente` modifica los objetos `Fuente` con `dimensionar()`. Si se crea una clase *Ventana Fuente* (`VFuente`) con herencia múltiple, se puede producir ambigüedad en el uso de `dimensionar()`.

```
class VFuente : public Ventana, public Fuente {...};
VFuente v;
v.dimensionar();    // se produce un error, ¿cuál?
```

La llamada a `dimensionar` es ambigua, ya que el compilador no sabrá a qué función `dimensionar` ha de llamar. Esta ambigüedad se resuelve fácilmente con el operador de resolución de ámbito (::).

```
v.Fuente::dimensionar();    // llamada a dimensionar de Fuente
v.Ventana::dimensionar();   // llamada a dimensionar de Ventana
```

Precaución

No es un error definir un objeto derivado con multiplicidad con ambigüedades. Éstas se consideran ambigüedades potenciales y sólo producen errores en tiempo de compilación cuando se llaman de modo ambiguo.

Regla

Incluso es mejor solución, que la citada anteriormente, resolver la ambigüedad en las propias definiciones de la función `dimensionar()`

```
class VFuente : public Ventana, public Fuente {
      ...
      void v_dimensionar() { Ventana::dimensionar(); }
      void f_dimensionar() { Fuente::dimensionar(); }
};
```

Ejemplo 14.8

```
class trabajador {
public:
   const int no_ss;
   const char* nombre;
   ...
};

class estudiante {
public:
   const char* nombre;
   ...
};

class estu_trabajador:public estudiante, public trabajador {
public:
   void imprimir() { cout << "número ss " << no_ss << endl;
   cout << nombre; ... }       // error
   ...
};
```

Para evitar error en la invocación a `nombre` se debe hacer uso del operador de resolución de ámbito.

```
estudiante::nombre
trabajador::nombre
```

14.4.2. Dominación (prioridad)

Un problema que se plantea cuando se manejan clases derivadas, es la *dominación* o *prioridad* que se produce cuando existen funciones en clases derivadas con el *mismo nombre* que otras funciones de las clases base. Consideremos el caso siguiente en herencia simple y luego extenderemos los conceptos a la herencia múltiple:

```
class Base {
public:
   void f(int);
   void f(char);
   void f(const String &);
};
```

```
class Derivada : public Base {
public:
   void f(const String &);      // oculta todas Base::f()
};

Derivada d;
d.f(5);     // error, Base::f(int) está oculta
d.f('x');   // error, Base::f(char) está oculta
d.f("abc"); // correcto, definida Derivada::f(const String &);
```

En herencia simple, las funciones en las clases derivadas con el *mismo nombre* que funciones de la clase base, *anulan* (reemplazan) a todas las versiones de la clase base. En el caso anterior, la función `Derivada::f()` oculta a todas las versiones de `Base::f()`. No se puede llamar a `f(int)` o a `f(char)` con un objeto de `Derivada` ya que `f()` de `Derivada` domina (tiene prioridad) a todas las `f()` de la clase `Base` incluso aunque sus signaturas sean distintas.

Con herencia múltiple, una clase derivada por multiplicidad puede heredar nombres de funciones equivalentes de dos clases base diferentes. Ninguna clase base *domina* (tiene prioridad) sobre la otra. Esta *ausencia de dominio* crea una ambigüedad, incluso con signaturas (prototipos) distintas.

Ejemplo

```
class A {
public:
   ...
   void f(int);
};

class B {
public:
   ...
   void f(const String &);
};

class C : public A, public B {...};

C c;
c.f(15);          // error, ambiguo
```

Cuando se llama a `f(int)` con un objeto `C`, el compilador informa de una ambigüedad, incluso aunque la signatura en `A` sea diferente de la de `B::f (const String &)`. El nombre de `f` es ambiguo, debido a que ninguna clase base domina sobre la otra. La resolución de la sobrecarga no se aplica a través del ámbito de la clase.

Regla

Cuando la dominación crea ambigüedades, se realizan llamadas directas en la clase `Derivada` a la respectiva clase `Base`

```
class C : public A, public B {
public:
   ...
   void f(int i) {A::f(); }
   void f(const String &s) {B::f(s); }
};
```

14.4.3. Inicialización de la clase base

¿Cómo llama el compilador a los constructores y destructores de las clases con herencia múltiple? Un ejemplo aclarará este problema.

```
class A {             // clase base
public:
   A(int);            // constructor con argumentos
   ˜A();              // destructor
};

class B {             // clase base
public:
   B();               // constructor sin argumentos
   ˜B();              // destructor
};

class C {             // clase base
public:
   C(double);         // constructor con argumentos
   ˜C();              // destructor
};

class D : public A, public B, public C { // herencia múltiple
public:
   D(int i, double m) :A(i), C(m) {}     // construcciones A,B,C y D
   ˜D();                                 // destruye D,C,B,A
};
```

La clase D deriva públicamente de las clases base A, B y C. Los constructores de A y C tienen un argumento cada una, mientras que B tiene un constructor void. Para construir objetos D, el constructor D pasa su argumento entero al constructor A y su argumento doble (double) al constructor de C. Obsérvese que la lista de inicialización de miembros no necesita pasar argumentos al constructor de B.

La *definición de la clase* determina el orden de las llamadas de constructores y destructores, no la lista de inicialización de miembros. Esto implica que las listas de inicialización de miembros puede inicializar clases base en *cualquier* orden. En otras palabras, el siguiente constructor modificado de D llama a todos los constructores base en el mismo orden que antes.

```
D(int i, double m) : C(m), A(i) {}   // construye A,B,C,D
```

Los cambios a la definición de la clase, por otra parte, afectan al orden de llamadas de constructores.

```
class D : public C, public B, public A {
public:
   D(int i, double m) : A(i), C(m) {}// construye C,B,A,D
   ˜D();                             // destruye D,A,B,C
};
```

Precaución

Si el orden de llamadas de los constructores base es importante en una declaración de herencia múltiple, asegúrese de que la lista de clases base es correcta.

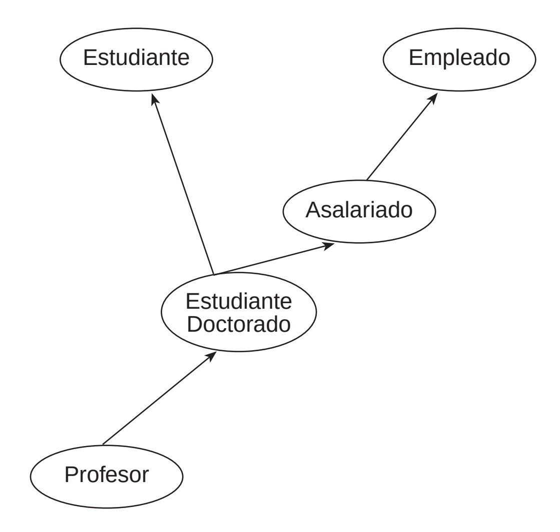
Regla

Orden de ejecución de constructores

1. Clases base inicializadas en orden de declaración.
2. Miembros inicializados en orden de declaración.
3. El cuerpo del constructor.

Ejemplo 14.9

Diseñar e implementar una jerarquía de clases que represente las relaciones entre las clases siguientes: estudiante, empleado, empleado asalariado y un estudiante de doctorado que es, a su vez, profesor de prácticas de laboratorio.



Nota: se deja la resolución como ejercicio al lector.

14.5. LIGADURA

Ligadura representa, generalmente, una conexión entre una entidad y sus propiedades. Si la propiedad se limita a funciones, ligadura es la conexión entre la llamada a función y el código que se ejecuta tras la llamada. Desde el punto de vista de atributos, la *ligadura* es el proceso de asociar un atributo a un nombre.

El momento en que un atributo o función se asocia con sus valores o funciones se denomina *tiempo de ligadura*. La ligadura se clasifica según sea el tiempo o momento de la ligadura: *estática* y *dinámica*. Ligadura estática se produce antes de la ejecución (durante la compilación), mientras que la ligadura dinámica ocurre durante la ejecución. Un atributo que se liga dinámicamente es un atributo dinámico. En un lenguaje de programación con ligadura estática, todas las referencias se determinan en tiempo de compilación. La mayoría de los lenguajes procedimentales son de ligadura estática; el compilador y el enlazador definen directamente la posición fija del código que se ha de ejecutar en cada llamada a la función.

La ligadura dinámica supone que el código a ejecutar en respuesta a un mensaje no se determinará hasta el momento de la ejecución. Únicamente la ejecución del programa (normalmente el valor de un puntero a una clase base) determinará la ligadura efectiva entre las diversas que son posibles (una para cada clase derivada).

La principal ventaja de la ligadura dinámica frente a la ligadura estática es que la ligadura dinámica ofrece un alto grado de flexibilidad y diversas ventajas prácticas y manejar jerarquías de clases de un modo muy simple. Entre las desventajas está que la ligadura dinámica es menos eficiente que la ligadura estática.

Los lenguajes orientados a objetos que siguen estrictamente el paradigma orientado a objetos ofrecen sólo ligadura dinámica. Los lenguajes que representan un compromiso entre el paradigma orientado a objeto y los lenguajes imperativos (tales como Simula y C++) ofrecen la posibilidad de elección con un tipo por defecto. En Simula y C++ la ligadura por defecto es estática y cuando se utiliza la declaración `virtual` se utiliza ligadura dinámica. En C++, siempre que se omite el especificador `virtual`, se supone que las referencias se resuelven en tiempo de compilación.

La ligadura en C++ es, por defecto, estática. La ligadura dinámica se produce cuando se hace preceder a la declaración de la función con la palabra reservada `virtual`. Sin embargo, puede darse el caso de ligadura estática, pese a utilizar `virtual`, a menos que el receptor se utilice como un puntero o como una referencia.

14.6. FUNCIONES VIRTUALES

Por omisión, las funciones C++ tienen ligadura estática; si la palabra reservada `virtual` precede a la declaración de una función, esta función se llama virtual, y le indica al compilador que puede ser definida (implementado su cuerpo) en una clase derivada y que en este caso la función se invocará directamente a través de un puntero. Se debe calificar una función miembro de una clase con la palabra reservada `virtual` sólo cuando exista una posibilidad de que otras clases puedan ser derivadas de aquélla.

Un uso común de las funciones virtuales es la declaración de clases abstractas y la implementación del polimorfismo.

Consideremos la clase `figura` como la clase base de la que se derivan otras clases, tales como `rectangulo,circulo` y `triangulo`. Cada figura debe tener la posibilidad de calcular su área y poder dibujarla. En este caso, la clase `figura` declara las siguientes funciones virtuales:

```
class figura {
public:
   virtual double calcular_area(void) const;
   virtual void dibujar(void) const;

   // otras funciones miembro que definen un interfaz a
   //todos los tipos de figuras geométricas
};
```

Cada clase derivada específica debe definir sus propias versiones concretas de las funciones que han sido declaradas virtuales en la clase base. Por consiguiente, si se derivan las clases `círculo` y `rectángulo` de la clase `figura`, se deben definir las funciones miembro `calcular_área` y dibujar en cada clase. Por ejemplo, las definiciones de la clase `círculo` pueden ser similares a éstas:

```
class circulo : public figura
{
   public:
      virtual double calcular_area(void) const;
      virtual void dibujar(void) const;
      // ...
   private:
      double xc, yc;        // coordenada del centro
      double radio;         // radio del círculo
};
```

```
#define PI 3.14159       // valor de "pi"
// Implementación de calcular_area
double circulo::calcular_area(void) const
{
   return(PI * radio * radio);
}
// Implementación de la función "dibujar"
void circulo::dibujar(void) const
{
   // ...
}
```

Cuando se declaran las funciones `dibujar` y `calcular_área` en la clase derivada, se puede añadir opcionalmente la palabra reservada `virtual` para destacar que estas funciones son verdaderamente virtuales. Las definiciones de las funciones no necesitan la palabra reservada `virtual`.

14.6.1. Ligadura dinámica mediante funciones virtuales

Las funciones virtuales se tratan igual que cualquier otra función miembro de una clase. Como ejemplo considérense las siguientes llamadas a las funciones virtuales `dibujar` y `calcular_área`:

```
círculo c1;
rectángulo r1;
double área = c1.calcular_área(); // calcular área del círculo
r1.dibujar();                     // dibujar un rectángulo
```

El uso anterior es similar al de cualquier función miembro. En este caso, el compilador C++ puede determinar que se desea llamar a la función `calcular_area` de la clase `círculo` y a la función `dibujar` de la clase `rectángulo`. De hecho, el compilador hace llamadas directas a estas funciones, y las llamadas a funciones se enlazan a un código específico en tiempo de enlace. Ésta es la ligadura que hemos denominado anteriormente *estática*.

Sin embargo, el caso más interesante es la llamada a las funciones a través de un puntero a `figura`, tal como:

```
figura* s[10];               // punteros a 10 objetos figuras
int i, numfiguras = 10;
// crea figuras y almacena punteros en array "s"
// dibujar las figuras
   for (i = 0; i < numfiguras; i++)
      figura[i] -> dibujar();
```

En este caso se produce ligadura dinámica; el compilador C++ no puede determinar cuál es la implementación específica de la función `dibujar()` que se ha de llamar.

Un puntero a una clase derivada es también un puntero a la clase base

En C++ se puede utilizar una referencia o un puntero a cualquier clase base, en lugar de una referencia o puntero a la clase derivada, sin una conversión explícita de tipos. De modo que si `circulo` y `rectangulo` se derivan de la clase `figura`, se puede llamar a una función que requiera un puntero a `figura` con un puntero a `circulo` o `rectangulo`.

Lo opuesto no es cierto; no se puede utilizar una referencia o un puntero a la clase derivada, en lugar de una referencia o un puntero a una instancia de una clase base.

El siguiente ejemplo muestra el uso de funciones virtuales y ligadura dinámica:

```
// archivo LIGADURA.CPP
// diferencias entre ligadura estática y dinámica

#include <iostream>
using namespace std;

// define una clase A con dos funciones que imprimen sus nombres

class A {
public:
   A() { }
   virtual void Dinamica()
   {
      cout << "Función dinámica de la clase A" << endl;
   }
   void Estatica()
   {
      cout << "Función estática de la clase A" << endl;
   }
};

// define una clase B, derivada de A, redefiniendo ambas funciones

class B : public A {
public:
   B() { }
   void Dinamica()
   {
      cout << "Función dinámica de clase B" << endl;
   }
   void Estatica()
   {
      cout << "Función estática de clase B" << endl;
   }
};

void main()
{
   // definiciones

   A *a;
   B *b;

   // inicialización

   a = new A();
   b = new B();

   cout << "Funciones del objeto a de la clase A" << endl;
   a -> Dinamica();
   a -> Estatica();
```

```
    cout << endl;
    cout << "Funciones del objeto b de la clase B" << endl;
    b -> Dinamica();
    b -> Estatica();
    cout << endl;

    // propiedad de polimorfismo
    // a toma como valor un puntero a un objeto de la clase B

    a = b;

    cout << "Funciones del objeto a de la clase A" << endl;
       << "al que se ha asignado un valor de la clase b" << endl;

    // llamada a la función virtual de la clase B

    a -> Dinamica();
    // llamada a la función no virtual de la clase B
    a -> Estatica();
}
```

Al ejecutar el programa se produce la siguiente salida:

```
Funciones del objeto a de la clase A
Función dinámica de la clase A
Función estática de la clase A

Funciones del objeto b de la clase B
Función dinámica de la clase B
Función estática de la clase B

Funciones del objeto a de la clase A
al que se ha asignado un valor de la clase B
Función dinámica de clase B
Función estática de clase A
```

En C++, las funciones virtuales siguen una regla concreta: la función se debe declarar como `virtual` en la primera clase en que está presente. Esta regla significa que, normalmente, las funciones virtuales se declaran en la clase de nivel más alto de una jerarquía.

14.7. POLIMORFISMO

En POO, el *polimorfismo* permite que diferentes objetos respondan de modo diferente al mismo mensaje. El polimorfismo adquiere su máxima potencia cuando se utiliza en unión de herencia.

Por ejemplo, si `figura` es una clase base de la que cada figura geométrica hereda características comunes, C++ permite que cada clase utilice una función (método) `Copiar` como nombre de una función miembro.

```
class figura {
    tipoenum tenum;          //tipoenum es un tipo enumerado
```

```
public:
   virtual void Copiar();
   virtual void Dibujar();
   virtual double Area);
};

class circulo : public figura {
   ...
public:
   void Copiar();
   void Dibujar();
   double Area();
};

clas rectangulo : public figura {
   ...
public:
   void Copiar(); // el polimorfismo permite que objetos diferentes
                  // tengan idénticos nombres de funciones miembro
   void Dibujar();
   double Area();
};
```

Otro ejemplo se puede apreciar en la jerarquía de clases:

```
class Poligono {     // superclase
public:
   float Perimetro();
   virtual float Area();
   virtual boolean PuntoInterior();
protected:
   void Visualizar();
};

class Rectangulo : public Poligono{

public:
   virtual float Area();
   virtual boolean PuntoInterior();
   void fijarRectangulo();
private:
   float Alto;
   float Bajo;
   float Izquierdo;
   float Derecho;
};
```

14.7.1. El polimorfismo sin ligadura dinámica

Como ya se ha comentado anteriormente, el polimorfismo permite que diferentes objetos respondan de modo diferente al mismo mensaje; por esta razón, en los programas se puede pasar el mismo mensaje a objetos diferentes, tales como:

```
switch(...) {
...
case Circulo:
   MiCirculo.Dibujar();
   d = MiCirculo.Area();
   break;
case Rectangulo:
   MiRectangulo.Dibujar();
   d = MiRectangulo.Area();
   break;
...
};
```

En el ejemplo anterior, cada figura recibe el mismo mensaje (por ejemplo, `MiCirculo.Dibujar`, `MiRectangulo.Area()`, etc.). Esta solución, sin embargo, aunque utiliza polimorfismo, no es aceptable, ya que impone las operaciones implementarias asociadas con el registro discriminante. Este código de discriminación se puede eliminar utilizando ligadura dinámica.

14.7.2. El polimorfismo con ligadura dinámica

Con ligadura dinámica, el tipo de objeto no es preciso decidirlo hasta el momento de la ejecución. El ejemplo de la sección anterior envía el programa al bloque apropiado de código basado en el tipo de objeto. Ejecutando esta sentencia `switch` en tiempo de ejecución, el programa modificará su flujo de ejecución dependiendo del valor del registro discriminante y su mantenimiento será difícil, ya que añadir objetos requerirá modificaciones a cada sentencia `switch` que haga uso del registro discriminante.

Una solución que hace uso de la ligadura dinámica puede ser ésta:

```
// crea e inicializa un array de figuras

figura *figuras[]= { new círculo, new rectángulo, new triángulo};
...
figuras[i].Dibujar();
```

Este segmento de código pasará el mensaje `Dibujar` a la figura apuntada por `figuras[i]`. La palabra clave `virtual` que se puso en la función `Dibujar` al declarar la clase base `Figura` ha indicado al compilador que esta función se puede llamar por un puntero. Mediante la ligadura dinámica, el programa determina el tipo de objeto en tiempo de ejecución, eliminando la necesidad del registro discriminante y la sentencia `switch` asociada.

El polimorfismo se puede representar con un array de elementos que se refieren a objetos de diferentes tipos (clases), como sugiere Meyer[1]. Así, en la Figura 14.4 se muestra un array que incluye punteros que apuntan a diferentes tipos, que son todos derivados de una superclase.

En un lenguaje de programación que no soporte polimorfismo, la estructura correspondiente se deberá implementar con un registro cuyos componentes se refieran a los diferentes tipos de puntos; las referencias serán estáticas, mientras que las referencias polimórficas serán dinámicas.

---

[1] Meyer, B., *Object-Oriented Software Construction*, New York, Prentice-Hall, 1998. (Esta obra fue traducida al español por un equipo de profesores universitarios coordinados y dirigidos por el autor de esta obra, que escribió también el prólogo a dicha edición española.)

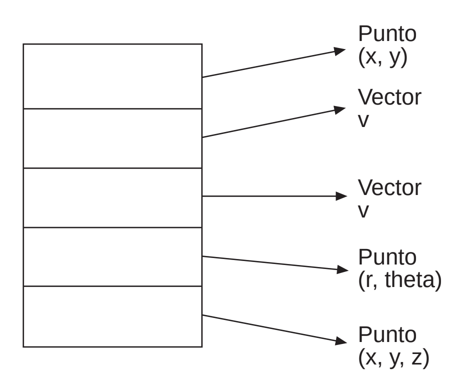


Figura 14.4. Referencias polimórficas.

14.8. USO DEL POLIMORFISMO

El polimorfismo permite utilizar el mismo interfaz, tal como métodos (funciones miembro) denominados `dibujar` y `calcular_area`, para trabajar con toda clase de figuras.

La forma más adecuada de usar polimorfismo es a través de punteros. Supongamos que se dispone de una colección de objetos `figura` en una estructura de datos, tal como un array o una lista enlazada. El array almacena simplemente punteros a objetos `figura`. Estos punteros apuntan a cualquier tipo de figura. Cuando se actúa sobre estas figuras, basta simplemente recorrer el array con un bucle e invocar a la función miembro apropiada mediante el puntero a la instancia. Naturalmente, para realizar esta tarea las funciones miembro deben ser declaradas como virtuales en la clase `figura`, que es la clase base de todas las figuras geométricas.

Para poder utilizar polimorfismo en C++ se deben seguir las siguientes reglas:

1. Crear una jerarquía de clases con las operaciones importantes definidas por las funciones miembro declaradas como virtuales en la clase base.
2. Las implementaciones específicas de las funciones virtuales se deben hacer en las clases derivadas. Cada clase derivada puede tener su propia versión de las funciones. Por ejemplo, la implementación de la función `dibujar` varía de una figura a otra.
3. Las instancias de estas clases se manipulan a través de una referencia o un puntero. Este mecanismo es la ligadura dinámica base.

Se obtiene ligadura dinámica sólo cuando las funciones miembro virtuales se invocan a través de un puntero a una instancia de una clase base.

14.9. LIGADURA DINÁMICA FRENTE A LIGADURA ESTÁTICA

Ligadura dinámica o postergada (tardía) se produce cuando una función polimórfica se define para clases diferentes de una familia pero el código real de la función no se conecta o enlaza hasta el tiempo de ejecución. Una función polimórfica que se enlaza dinámicamente se llama **función virtual**.

La ligadura dinámica se implementa en C++ mediante *funciones virtuales*. Con ligadura dinámica, la selección del código a ejecutar cuando se llama a una función virtual se retrasa hasta el tiempo de ejecución. Esto significa que cuando se llama a una función virtual, el código ejecutable determina en tiempo de ejecución cuál es la versión de la función que se llama. Recordemos que las funciones virtuales son polimórficas y, por consiguiente, tienen diferentes implementaciones para clases diferentes de la familia.

La ligadura estática se produce cuando se define una función polimórfica para diferentes clases de una familia y el código real de la función se conecta o enlaza en tiempo de compilación. Las funciones sobrecargadas se enlazan estáticamente.

La ligadura estática se produce cuando el código de la función «se enlaza» en tiempo de compilación. Esto significa que cuando se llama una función no virtual, el compilador determina en tiempo de compilación cuál es la versión de la función a llamar. Las funciones sobrecargadas se enlazan estáticamente, mientras que las funciones virtuales se enlazan dinámicamente. Con funciones sobrecargadas, el compilador puede determinar cuál es la función a llamar basada en el número y tipos de datos de los parámetros de función. Sin embargo, las funciones virtuales tienen la misma interfaz dentro de una familia de clases dada. Por consiguiente, los punteros se deben utilizar durante el tiempo de ejecución para determinar cuál es la función a llamar.

Las funciones virtuales se declaran en una clase base en C++ utilizando la palabra reservada **`virtual`**. Cuando se declara una función como una función virtual de una clase base, el compilador conoce cuál es la definición de la clase base que se puede anular en una clase derivada. La definición de la clase base se anula (reemplaza) definiendo una implementación diferente para la misma función de la clase derivada. Si la definición de la clase base no se anula en una clase derivada, entonces la definición de la clase base está disponible a la clase derivada.

14.10. VENTAJAS DEL POLIMORFISMO

El polimorfismo hace su sistema más flexible, sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación. Éste es el caso de C++.

Sin embargo, otros lenguajes de programación orientada a objetos soportan un polimorfismo ilimitado que choca con la idea de la comprobación de tipos. Aunque ofrecen más flexibilidad y libertad, tienden a evitar casi cualquier comprobación del código fuente, postergándolo al momento de la ejecución. Se tiene más libertad para realizar la programación, pero se tiene más probabilidad de errores potenciales. Como resultado, el código producido con otros lenguajes orientados a objetos es menos seguro, debido a que las llamadas a funciones que se encuentran no se pueden definir para algunos objetos, por razones que van desde un simple error de sintaxis a un error conceptual o lógico (el programa resultante tiende a ser más lento, ya que muchas comprobaciones que se necesitan se hacen en tiempo de ejecución). En el lenguaje Smalltalk (que ignora la comprobación de tipos estática), un mensaje puede no tener un método correspondiente, pero el entorno atrapará el error sólo cuando se intenta ejecutar la instrucción defectuosa.

Aunque existe menos libertad, el polimorfismo en C++ es una herramienta muy potente y que puede ser utilizada en muchas situaciones diferentes. Las aplicaciones más frecuentes del polimorfismo son:

- **Especialización de clases derivadas**. El uso más común del polimorfismo es derivar clases especializadas de clases que han sido definidas. Así por ejemplo, una clase `cuadrado` es una especialización de la clase `rectangulo` (cualquier cuadrado es un tipo de rectángulo). Esta clase de polimorfismo aumenta la eficiencia de la subclase, mientras conserva un alto grado de flexibilidad y permite un medio uniforme de manejar rectángulos y cuadrados.

- **Estructuras de datos heterogéneos**. A veces es muy útil poder manipular conjuntos similares de objetos. Con polimorfismo se pueden crear y manejar fácilmente estructuras de datos heterogéneos, que son fáciles de diseñar y dibujar, sin perder la comprobación de tipos de los elementos utilizados.

- **Gestión de una jerarquía de clases**. Las jerarquías de clases son colecciones de clases altamente estructuradas, con relaciones de herencia que se pueden extender fácilmente.

RESUMEN

La relación ***es-un*** indica herencia. Por ejemplo, una rosa es un tipo de flor; un pastor alemán es un tipo de perro, etc. La relación ***es-un*** es transitiva. Un pastor alemán es un tipo de perro y un perro es un mamífero; por consiguiente, un pastor alemán es un mamífero. La relación ***tiene-un*** indica contenido. Por ejemplo, una radio tiene sintonizador. Un coche (carro) tiene un motor.

Una clase nueva que se crea a partir de una clase ya existente, utilizando herencia, se denomina *clase derivada* o *subclase.* La clase padre se denomina *clase base* o *superclase.*

Herencia simple es la relación entre clases que se produce cuando una nueva clase se crea utilizando las propiedades de una clase ya existente. La nueva clase se denomina clase base. Las relaciones de herencia reducen código redundante en programas. Uno de los requisitos para que un lenguaje sea considerado orientado a objetos o basado en objetos es que soporte herencia.

Herencia privada es el término que se utiliza cuando una clase derivada restringe el acceso a los miembros de su clase haciéndolos privados. Esto impide a los usuarios de la clase derivada acceder a los miembros de la clase base.

Herencia múltiple, se produce cuando una clase se deriva de dos o más clases base. Aunque es una herramienta potente, puede crear problemas, especialmente de colisión o conflicto de nombres, cosa que se produce cuando nombres idénticos aparecen en más de una clase base.

Poliformismo es la propiedad de que «algo» tome diferentes formas. En un lenguaje orientado a objetos el poliformismo es la propiedad por la que un mensaje puede significar cosas diferentes dependiendo del objeto que lo recibe.

Para implementar el poliformismo, un lenguaje debe soportar ligadura dinámica. La razón por que el poliformismo es útil es que proporciona la capacidad de manipular instancias de clases derivadas a través de un conjunto de operaciones definidas en su clase base. Cada clase derivada puede implementar las operaciones definidas en la clase base.

Una clase abstracta es aquella que contiene al menos una función virtual pura. Una función virtual pura es una función miembro que se declara utilizando un especificador puro que significa que el prototipo de la función termina con un «= 0». La función se debe implementar en cualquier clase derivada antes de que se creen las instancias.

EJERCICIOS

14.1. Definir una clase base `persona` que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.). Diseñar una jerarquía de clases que contemple las clases siguientes: `estudiante`, `empleado`, `estudiante_empleado`.

Escribir un programa que lea un archivo de información y cree una lista de personas: *a*) general; *b*) estudiantes; *c*) empleados; *d*) estudiantes empleados. El programa debe permitir ordenar alfabéticamente por el primer apellido.

14.2. Implementar una jerarquía `Libreria` que tenga al menos una docena de clases. Considérese una *librería* que tenga colecciones de libros de literatura, humanidades, tecnología, etc.

14.3. Diseñar una jerarquía de clases que utilice como clase base o raíz una clase LAN (red de área local).

Las subclases derivadas deben representar diferentes topologías, como *estrella, anillo, bus* y *hub*. Los miembros datos deben representar propiedades tales como *soporte de transmisión, control de acceso, formato del marco de datos, estándares, velocidad de transmisión*, etc. **Se desea simular la actividad de los nodos de tal LAN.**

La red consta de **nodos**, que pueden ser dispositivos tales como computadoras personales, estaciones de trabajo, máquinas FAX, etc. Una tarea principal de LAN es soportar comunicaciones de datos entre sus nodos. El usuario del proceso de simulación debe, como mínimo, poder:

- Enumerar los nodos actuales de la red LAN.
- Añadir un nuevo nodo a la red LAN.
- Quitar un nodo de la red LAN.
- Configurar la red, proporcionándole una topología de *estrella* o en *bus*.

- Especificar el tamaño del paquete, que es el tamaño en bytes del mensaje que va de un nodo a otro.
- Enviar un paquete de un nodo especificado a otro.
- Difundir un paquete desde un nodo a todos los demás de la red.
- Realizar estadísticas de la LAN, tales como tiempo medio que emplea un paquete.

14.4. Implementar una jerarquía `Empleado` de cualquier tipo de empresa que le sea famililiar. La jerarquía debe tener al menos cuatro niveles, con herencia de miembros dato, y métodos. Los métodos deben poder calcular salarios, despidos, promoción, dar de alta, jubilación, etc. Los métodos deben permitir también calcular aumentos salariales y primas para `Empleados` de acuerdo con su categoría y productividad. La jerarquía de herencia debe poder ser utilizada para proporcionar diferentes tipos de acceso a `Empleados`. Por ejemplo, el tipo de acceso garantizado al público diferirá del tipo de acceso proporcionado a un supervisor de empleado, al departamento de nóminas, o al Ministerio de Hacienda. Utilice la herencia para distinguir entre al menos cuatro tipos diferentes de acceso a la información de `Empleado`.

14.5. Implementar una clase `Automóvil` (*Carro*) dentro de una jerarquía de herencia múltiple. Considere que, además de ser un *Vehículo*, un automóvil es también una *comodidad, un símbolo de estado social, un modo de transporte*, etcétera. `Automóvil` debe tener al menos tres clases base y al menos tres clases derivadas.

14.6. Escribir una clase `FigGeométrica` que represente figuras geométricas tales como *punto, línea, rectángulo, triángulo* y similares. Debe proporcionar métodos que permitan dibujar, ampliar, mover y destruir tales objetos. La jerarquía debe constar al menos de una docena de clases.

14.7. Implementar una jerarquía de tipos datos numéricos que extienda los tipos de datos fundamentales tales como `int` y `float`, disponibles en C++. Las clases a diseñar pueden ser `Complejo`, `Fracción`, `Vector`, `Matriz`, etc.

14.8. Implementar una jerarquía de herencia de animales tal que contenga al menos seis niveles de derivación y doce clases.

14.9. Diseñar la siguiente jerarquía de clases:

Persona
Nombre
edad
visualizar()

Estudiante		*Profesor*	
nombre	heredado	nombre	heredado
edad	heredado	edad	heredado
id	definido	salario	definido
visualizar()	*redefinido*	visualizar	*heredada*

Escribir un programa que manipule la jerarquía de clases, lea un objeto de cada clase y lo visualice.

a) Sin utilizar funciones virtuales.
b) Utilizando funciones virtuales.

14.10. El archivo `VIRTUAL.CPP` muestra las diferencias entre llamadas a una función virtual y a una función normal.

```
// archivo VIRTUAL.CPP
#include <iostream>

class Base {
public:
   virtual void f() { cout << "f()
      : clase base-!" << endl;
   void g() {cout << "g():clase
      base-!" << endl;)
};

class Derivada1:public Base {
public:
   virtual void f() {cout <<"f():
      clase Derivada-!"
   << endl;)
   void g() {cout << "g() :clase
      Derivada-!" << endl;}
};

class derivada2 : public Derivada1 {
public:
   virtual void f() {cout << "f() :
      clase Derivada2-!"endl;}
   void g() {cout << "g() :clase
      Derivada2-!" << endl;}
};

void main()
{
   Base b;
   Derivada1 d1;
   Derivada2 d2;
```

```
    base *p = &b;
    p ->f();
    p -> g();

    p= &d1;
    p ->f()
    p ->g();

    p=&d2;
    p ->f();
    p ->g();
}
```

¿Cuál es el resultado de ejecutar este programa? ¿Por qué?

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 288).

2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

14.1. ¿Cuál es la salida del siguiente programa que contiene funciones virtuales y no virtuales en la clase base?

```
#include <cstdlib>
#include <iostream>
using namespace std;

class base
{
  public:
      virtual void f(){ cout <<
      "f(): clase base " << endl;}
      void g() {cout << "g(): clase
      base " << endl;}
};
class Derivada1 : public base
{
   public:
       virtual void f(){cout <<
       "f(): clase Derivada 1 " <<
       endl;}
       void g() {cout << "g(): cla-
       se Derivada 1!" << endl;}
};
class Derivada2 : public Derivada1
{
   public:
      void f() {cout << "f(): clase
      Derivada 2 !" <<endl;}
      void g() {cout << "g(): clase
      Derivada 2 !" << endl;}
};

int main(int argc, char *argv[])
{
      base b;
      Derivada1 d1;
      Derivada2 d2;
      base *p;
      p = &b;   p ->f();  p -> g();
      p = &d1;  p ->f();  p ->g();
      p=&d2;    p ->f();  p ->g();
      system("PAUSE");
      return EXIT_SUCCESS;
}
```

14.2. Declarar las clases `Punto unidimensional`, `Punto bidimensional` y punto tridimensional, que permitan tratar puntos en el espacio de dimensión uno, dos y tres respectivamente, mediante una jerarquía de clases.

14.3. Declarar la clase abstracta `Figura` con las funciones virtuales `Área` y `perímetro` y que tenga como clases derivadas las figuras geométricas `circulo`, `cuadrado` y `rectangulo`.

14.4. Diseñar una jerarquía de clases con la clase base `Deportes`, de la que se derivan las clases `Futbol`, `Baloncesto`, `Volley_Ball`, que tenga una ligadura dinámica.

CAPÍTULO 15

Genericidad: plantillas *(templates)*

Contenido

INTRODUCCIÓN

Una de las ideas claves en el mundo de la programación es la posibilidad de diseñar clases y funciones que actúen sobre tipos arbitrarios o genéricos. Para definir clases y funciones que operan sobre tipos arbitrarios se definen los *tipos parametrizado* o *tipos genéricos*. La mayoría de los lenguajes de programación orientados a objetos proporcionan soporte para la genericidad: las unidades genéricas (paquetes o funciones) en Ada, las *plantillas* (*templates*) en C++. C++ proporciona la característica **plantilla** (*template*), que permite a los tipos ser parámetros de clases y funciones. C++ soporta esta propiedad a partir de la versión 2.1 de AT&T.

Las *plantillas* o *tipos parametrizados* se pueden utilizar para implementar estructuras y algoritmos que son en su mayoría independientes del tipo de objetos sobre los que operan. Por ejemplo, una plantilla `Pila` puede describir cómo implementar una pila de objetos arbitrarios; una vez que la plantilla se ha definido, los usuarios pueden escribir el código que utiliza pilas de tipos de datos reales, cadenas, enteros, punteros, etc.

CONCEPTOS CLAVE

- Concepto de tipo genérico.
- Conceptos fundamentales de plantillas en C++.
- Funciones de plantilla.
- Generacidad.
- Plantillas de funciones.
- Plantillas frente a polimorfismo.
- Tipo parametrizado.
- `template`.

15.1. GENERICIDAD

La genericidad es una construcción importante en un lenguaje de programación orientada a objetos, que si bien no es exclusivo de este tipo de lenguajes, es en ellos en los que ha adquirido verdadera carta de naturaleza, dado que ha servido principalmente para aumentar la reutilización.

La genericidad es una propiedad que permite definir una clase (o una función) sin especificar el tipo de datos de uno o más de sus miembros (parámetros). De esta forma, se puede cambiar la clase para adaptarla a los diferentes usos sin tener que rescribirla.

La razón de la genericidad se basa principalmente en el hecho de que los algoritmos de resolución de numerosos problemas no dependen del tipo de datos que procesa y, sin embargo, cuando se implementan en un lenguaje de programación, los programas que resuelven cada algoritmo serán diferentes para cada tipo de dato que procesan. Por ejemplo, un algoritmo que implementa una pila de caracteres es esencialmente el mismo que el algoritmo necesario para implementar una pila de enteros o de cualquier otro tipo. Así, por ejemplo, en Pascal, C, COBOL, etc., se requiere un programa distinto para manejar una pila de enteros, de reales, de cadenas, etc. Sin embargo, en C++ (a partir de la versión 3.0 de AT&T) y en Java existen las plantillas (*templates*); en Ada, los paquetes genéricos, etc., que permiten definir unas *clases genéricas* o *paramétricas* que pueden implementar esas estructuras o clases con independencia del tipo de elemento que procesan. Es decir, las plantillas, paquetes, etc., pueden diseñar pilas de elementos con independencia del tipo de elemento que procesan. Las plantillas o unidades genéricas, tras ser implementadas, han de ser instanciadas para producir subprogramas, paquetes o clases reales que ya utilizan tipos de datos concretos. Las clases genéricas se denominan también **clases contenedoras** (*container class*), clases que contienen objetos de algún otro tipo. Ejemplos típicos de clases contenedoras son *pilas, colas, listas, conjuntos, diccionarios, arrays*, etc. Estas clases se definen con independencia del tipo de los objetos contenidos, y es el usuario de la clase quien deberá especificar el tipo de argumento de la clase en el momento que se instancia.

La genericidad ha sido definida de diversas formas; en nuestro caso hemos seleccionado la definición dada por Bertrand Meyer[1] (autor del lenguaje Eiffel).

En el caso más común, los parámetros representan tipos. Los módulos reales, denominados instancias del módulo genérico, se obtienen proporcionando tipos reales para cada uno de los parámetros genéricos.

Booch señala cuatro métodos para construir clases contenedoras:

- Usar macros —en algunos lenguajes— para compilar una clase varias veces, cada vez que se proporciona una definición del tipo que realmente se utiliza.
- Otros lenguajes, tales como Smalltalk, evitan comprobación de tipos. En Smalltalk, el polimorfismo no se limita a jerarquías, como en C++, y cualquier clase puede ser sustituida por otra. Por consiguiente, cualquier contenedor es totalmente heterogéneo.
- Este enfoque puede ser ligeramente modificado añadiendo pruebas a los contenedores polimórficos.
- El último enfoque es el uso de módulos genéricos. Algunos lenguajes proporcionan un mecanismo para clases parametrizadas, tales como en los casos de Ada, Eiffel y C++.

La genericidad se implementa en Ada mediante unidades genéricas y en C++ con plantilla de clases y plantillas de funciones.

15.2. CONCEPTOS FUNDAMENTALES DE PLANTILLAS EN C++

Las **plantillas** (*templates*) fueron propuestas por Stroustrup[2] en la conferencia USENIX C++ de Denver, en 1988. En esa ocasión planteó las siguientes preguntas:

- ¿Puede ser fácil de utilizar la parametrización de tipos?

[1] *Genericidad* es la capacidad de definir módulos parametrizados. Tal módulo, denominado módulo genérico, no es directamente útil; más bien es un patrón de módulos.

[2] Stroustrup, Bjarne, *Parametrized Types for C++*. Proceeding Acts, USENIX C++ Conference, Denver, octubre 1988.

- ¿Pueden objetos de tipos parametrizados ser integrados en C++?
- ¿Pueden los tipos parametrizados ser implementados de modo que la velocidad de compilación y enlace sea similar al realizado por un sistema de compilación que no soporta parametrización de tipos?
- ¿Puede tal sistema de compilación ser simple y transportable?

Naturalmente, el mismo Stroustrup comentaba en la citada conferencia que su respuesta a todas esas preguntas era *sí*. El mecanismo de plantillas presentado en el ARM (*Annotated Reference Manual*) en julio de 1990 fue aceptado por el comité ANSI C++. Por fin, la versión 3.0 del C++ de AT&T introdujo la noción de plantilla (*template*), una construcción que permite escribir funciones y clases muy generales que pueden aplicarse a objetos de tipos diferentes. Existen dos tipos de plantillas: *plantillas de clases* y *plantillas de funciones*.

El interés de las plantillas proviene del hecho de que esta generalidad no arrastra pérdida de rendimiento y no obliga a sacrificar las ventajas de C++ en materia de control estricto de los tipos de datos.

La traducción de **template** en esta obra ha sido **plantilla**[3], aunque también puede utilizarse *patrones*, *modelos,* o también *tipos genéricos* o *tipos parametrizados*. Asimismo, utilizaremos indistintamente los términos *plantilla de funciones* y *funciones plantillas,* o también *plantilla de clases* o *clases plantillas*.

Una *plantilla* es un patrón para creación de clases o funciones como *instancias* o *especializaciones* de la plantilla en tiempo de compilación, de igual forma que una clase es un patrón para crear objetos como instancias de la clase en tiempo de ejecución. Por esta razón, se denominan a veces *función patrón* y *clase patrón* en lugar de plantilla de funciones y plantilla de clases, como se verá más tarde.

En los lenguajes de programación es muy frecuente que las funciones que realizan una determinada tarea, deban ser definidas repetidamente a medida que los tipos de los parámetros son diferentes. Ejemplos típicos son las funciones `maximo/minimo` que devuelven en el máximo o `minimo` de dos valores `enteros`; `intercambiar` función que intercambia entre sí los valores `enteros` de dos variables.

Estas funciones se deben volver a implementar para cada pareja de tipos para las cuales se necesite el máximo, mínimo o el intercambio de valores.

Ejemplo

```
//esta función se aplica a tipos enteros
void Intercambio (int& m, int& n)
{
   int aux,
   aux = m;
   m = n;
   n = aux;
}
```

Si alguna se desea intercambiar valores de tipo `char`, se puede sobrecargar la función `Intercambio` con la siguiente definición:

```
void Intercambio (char& var, char& var2)
{
   char aux;

   aux = var1;
```

[3] Este término es el que hemos utilizado en nuestras clases, cursos y conferencias impartidas en España y Latinoamérica en los últimos años. No obstante, hemos comprobado personalmente que también se utiliza en Latinoamérica el término *patrón* (este término es el empleado en la traducción del libro *El lenguaje de programación C++*, de Stroustrup, realizada en México).

```
   var1 = var2;
   var2 = aux;
}
```

Si ahora se desea utilizar la función `Intercambio` para aplicarla a pares de variables tipo `double` o `float`, se tendrá que escribir de nuevo una tercera y una cuarta definición de la función casi idéntica a las anteriores.

Igual sucedería si consideramos la función `maximo` de dos valores, o cualquier otra función. Se requiere mucho esfuerzo de implementación repetida de funciones. Por esta razón, sería importante diferenciar de una característica especial del lenguaje cuya definición de función se pudiera aplicar a variables de cualquier tipo, con una sintaxis similar a

```
void Intercambio (tipo_variable& var1, tipo_variable& var2)
{
   tipo_variable aux;

   aux = var1;
   var1 = var2;
   var2 = aux;
}
```

Esta característica se dispone en C++ y permite definir funciones que se apliquen a todos los tipos de variables, aunque la sintaxis a utilizar será un poco más complicada. La entidad que permite estas características se denomina **plantillas** (*templates*).

Las plantillas o patrones son funciones y clases que no están implementadas para un tipo determinado, sino para un tipo que se debe definir en cada momento. Para utilizar estas funciones o clases el programador sólo debe especificar el tipo para el cual debe realizarse la plantilla. Así la función `maximo` o `Intercambio` sólo deben definirse e implementarse una vez. Igual sucede con clases contenedoras tales como pila, lista enlazadas, etc., que sólo deben ser implementada una vez.

Terminología

Los términos utilizados para describir plantillas no están definidos con rigurosidad. Así, una función cuyo tipo está parametrizado se le conoce a veces como *plantilla de funciones*[4] (*function template*) y a veces como *función plantilla*. En general, nosotros la definiremos como *plantilla de funciones*. De modo similar para las clases con tipos parametrizados, utilizaremos *plantillas de clases* en lugar de *clases plantilla* o *clases patrón*.

El proceso de creación de una clase función o función de miembro de una plantilla mediante la sustitución real de los valores sus argumentos se llama *instanciación* de la plantilla. Recuerde que la palabra *instanciar* se utiliza en programación orientada a objetos para la creación de los objetos de una clase. Por consiguiente, el término *instanciar* en C++ dependerá del contexto en que se utiliza. Una *instancia*, *ejemplar* o *especialización* de una plantilla función crea una función; una instancia de una plantilla de clases crea una clase y todos sus miembros.

15.3. PLANTILLAS DE FUNCIONES

Una *plantilla de funciones* especifica un conjunto infinito de funciones sobrecargadas. Cada función de este conjunto es una *función plantilla* y una instancia de la plantilla de función. Una función plantilla apropiada se produce automáticamente por el compilador cuando sea necesario.

[4] También se utilizan los términos *plantilla patrón*, o lo que es igual *función patrón* o *clase patrón*.

Una plantilla de funciones se utiliza para definir un grupo de funciones que se pueden utilizar para tipos diferentes. Una plantilla de funciones es un conjunto indeterminado de funciones sobrecargadas y describe las propiedades específicas de una función. Se puede sobrecargar una plantilla de funciones con una función no plantilla o con otras plantillas de funciones. Se puede, incluso, disponer de una plantilla de función y una función no plantilla con el mismo nombre y parámetros.

15.3.1. Fundamentos teóricos

Supongamos que se desea escribir una función `min(a, b)` que devuelve el valor más pequeño de sus argumentos.

C++ impone una declaración precisa de los tipos de argumentos necesarios que recibe `min()`, así como el tipo de valor devuelto por `min()`. Es necesario utilizar diferentes funciones sobrecargadas `min()`, cada una de las cuales se aplica a un tipo de argumento específico. Un programa que hace uso de funciones `min()` es:

```
// archivo PLANFUN.CPP

#include <iostream>
using namespace std;

// datos enteros (int)
int min(int a, int b)
{
   if(a <= b)
      return a;
   return b;
}

// datos largos

long min(long a, long b)
{
   if(a <= b)
      return a;
   return b;
}

// datos char

char min(char a, char b)
{
   if(a <= b)
      return a;
   return b;
}

// datos double
```

```
double min(double a, double b)
{
   if(a <= b)
      return a;
   return b;
}

int main()
{
   int ea = 1, eb = 5;
   cout << "(int):" << min(ea, eb) << endl;

   long la = 10000, 1b = 4000;
   cout << "(long):" << min(1a, 1b) << endl;

   char ca = 'a', cb = 'z';
   cout << "(char):" << min(ca, cb) << endl;

   double da = 423.654, db = 789.10;
   cout << "(double):" << min(da, db) << endl;
}
```

Al ejecutar el programa se visualiza:

```
(int) : 1
(long): 4000
(char) : a
(double): 423.654
```

Obsérvese, que las diferentes funciones `min()` tienen un cuerpo idéntico, pero como los argumentos son diferentes, se diferencian entre sí en los prototipos. En este caso sencillo se podría evitar esta multiplicidad de funciones definiendo una macro con `#define`:

```
#define min(a, b) ((a) <= (b) ? (a) : (b))
```

Sin embargo, con la macro se perderán los beneficios de las verificaciones de tipos que efectúa C++ para evitar errores. Para seguir disponiendo de las ventajas de las verificaciones de tipos se requieren las plantillas de funciones.

Las funciones `min` anteriores se escriben igual en todos los casos, pero son funciones totalmente diferentes ya que ellas manejan argumentos y valores de retorno de tipos diferentes. Es cierto, que en C++ estas funciones están sobrecargadas con el mismo nombre, pero se deben escribir definiciones independientes de cada una de ellas, en el lenguaje C que no soporta la sobrecarga de funciones para tipos diferentes no pueden tener el mismo nombre, de modo que en C++ una función `abs` puede estar sobrecargada mientras que la biblioteca C soporta funciones tales como `abs()`, `fabs()`, `labs()` y `cabs()` para representar diferentes funciones con tipos diferentes.

La reescritura del código del cuerpo de la función consume tiempo y malgasta espacio en el código y luego en memoria. También, caso de encontrar un error en alguna función, se necesitará corregir ese error en el cuerpo de todas las funciones. La plantilla de funciones permite escribir una función patrón una sola vez y utilizarla en muchos tipos diferentes. La idea se muestra en la Figura 15.1.

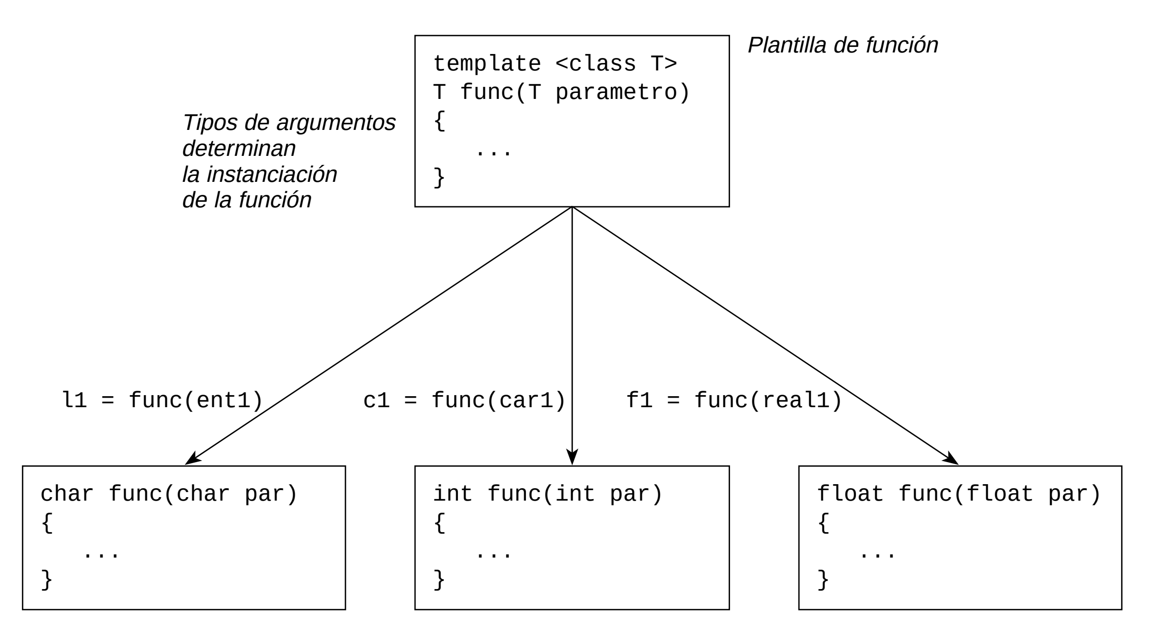


Figura 15.1. Plantilla de función.

15.3.2. Definición de plantilla de funciones

La innovación clave en las plantillas de funciones es representar el tipo de dato utilizado por la función no como un tipo específico tal como `int`, sino por un nombre que representa a cualquier tipo. Normalmente, este tipo se representa por `T` (aunque puede ser cualquier nombre que decida el programador, tal como `Tipo`, `UnTipo`, `Complejo` o similar; es decir, cualquier identificador distinto de una palabra reservada).

La sintaxis de una plantilla de funciones tiene dos formatos, según se utilice la palabra reservada `class` o `typename`. Ambos formatos se pueden utilizar: `class`, es el formato clásico y que incorpora todos los compiladores y, `typename`, es el formato introducido por el estándar ANSI/ISO C++ para utilizar en lugar de `class`.

Sintaxis

```
1. template <class T>
2. template <typename T>
```

T es un parámetro tipo, que puede ser reemplazado por cualquier tipo, y que también se conoce como un argumento de la plantilla.

Sintacticamente, no hay ninguna diferencia entre las dos palabras reservadas, `class` y `typename`, y se pueden usar, una u otra, indistintamente.

Una definición de plantilla comienza con la palabra reservada `template` seguida por una *lista de parámetros de la plantilla*, que es una lista de uno o más parámetros de plantilla, separados por comas, encerrados entre corchetes tipo ángulo (< y >). *La lista de parámetros no puede estar vacía.*

Reglas prácticas

- La palabra reservada `class` se puede utilizar en lugar de `typename` y tienen el mismo significado en este contexto (una recomendación puede ser: utilizar `class` cuando el agumento deba ser una clase y `typename` cuando el argumento pueda ser cualquier tipo).
- La palabra reservada `template` indica al compilador que el código que sigue, es una plantilla o patrón de funciones, no la cabecera o definición real de una función.
- Los parámetros de tipo (y los argumentos para las plantillas de clase) aparecen entre corchetes de desigualdad (<>).
- Las plantillas de funciones, al contrario que las funciones ordinarias, no se pueden separar en un archivo de cabecera que tenga su cabecera y un archivo compilado por separado que contenga sus definiciones. Las definiciones se deben compilar con las cabeceras que se deben incluir en un programa mediante `#include`.

Las cabeceras de las plantillas de funciones no se pueden guardar en un archivo de cabecera `nombre.h` y las definiciones en un archivo de cabecera `nombre.ccp` que se compilen de modo separado. Una práctica habitual es poner todo en el mismo archivo.

Las plantillas de funciones pueden tener más de un parámetro de tipo.

Ejemplos

1.
```
//Devolver el valor mayor de dos valores
template <typename T>
const T& max(const T& a, const T& b)
{
   return a > b ? a : b;
}
```

2.
```
//Devolver el valor mínimo de a y b
template <typename T>
T min(T a, T b)
{return a < b ? a : b;}
```

Para utilizar una plantilla de funciones, se debe especificar una instancia de una plantilla, proporcionando argumentos para cada parámetro de la plantilla. Se puede hacer esta tarea, explícitamente, listando argumentos dentro de los corchetes tipo ángulo. Unos ejemplos de una llamada a la función `max` definida antes:

a. `long x = max <long> (40, 50);`
b. `int x = max (40, 50);`

Ejercicio 15.1

Mostrar el funcionamiento completo de una plantilla de función que realiza el intercambio de los valores de dos variables.

```
#include <iostream>
using name std;

//definir una plantilla de función, intercambio
template <class T>
```

```
void intercambio (T& v1, T& v2)
{
   T aux;

   aux = v1;
   v1 = v2;
   v2 = aux;
}
int main()
{
   int numero1 = 5, numero2 = 8;
   cont << "valores originales:"
        << numero1 << " " << numero2 << endl;
   intercambio (numero1, numero2);
   cout << "valores intercambiados: "
        << numero1 << " " << numero2 << endl;

   //intercambio de caracteres
   char car1 = 'L' , car2 = 'J' ;
   cout << "valores especiales: "
        << car1 << " " << car2 << endl;
   intercambio (car1, car2)
   cout << "valores intercambiados: "
        << car1 << " " << car2 << endl;
   return 0;
}
```

Ejercicio 15.2

Escribir una función para comparar dos valores e indicar si el primer valor es menor, igual o mayor que el segundo.

El sistema de sobrecarga de funciones puede definir varias funciones sobrecargadas:

```
//devolver 0 si los valores son iguales, -1 si v1 es el más
//pequeño y 1 si v2 es el menor.
int comparar(const string &v1, cons string &v2)
{
   if (v1 < v2) return -1;
   if (v2 < v1) return 1;
   return 0;
}
//otra función comparar puede ser
int comparar (const double &v1, const double &v2)
{
   if (v1 < v2) return -1;
   if (v2 < v1) return 1;
   return 0;
}
```

Estas dos funciones son casi idénticas excepto los tipos de sus parámetros que son diferentes: están *sobrecargadas*. Una versión de plantilla de función de `comparar` es:

```
template <typename T>
int comparar (const T &v1, const T &v2)
{
   if (v1 < v2) return -1;
   if (v2 < v1) return 1;
   return 0;
}
```

Al ejecutarse el programa se obtendrá:

```
valores originados : 5 8
valores intercambiados : 8 5
valores originales : L J
valores itercambiados J L
```

Plantilla de función `class`

Cada parámetro formal consta de la palabra reservada `class`, seguida por un identificador. Esta palabra reservada indica al compilador que el parámetro representa un posible tipo incorporado definido por el usuario. Se necesita, al menos, un parámetro `T` que proporcione datos sobre los que pueda operar la función. También se puede especificar un puntero (`T * parámetro`) o una referencia (`T & parámetro`).

La función puede declararse con un parámetro formal o con múltiples parámetros formales y devolver, inclusive, un valor de tipo `T`. Algunas posibles declaraciones pueden ser:

```
1. template <class T> T & f(T parámetro)
   {
      // cuerpo de la función
   }
2. template <class T> T f(int a, T b)
   {
      // cuerpo de la función
   }
3. template <class T> T f(T a, T b)
   {
      // cuerpo de la función
   }
```

Se pueden declarar también dos parámetros tipo `T`, pero con la condición de que sean distintos.

```
4. template <class T1, class T2> T1 f(T1 a, T2 b)
   {
      // cuerpo de la función
   }
```

Una función plantilla se puede declarar externa (`extern`), en línea (`inline`) o estática (`static`), de igual forma que una función no plantilla. El especificar correspondiente se sitúa a continuación de la línea de parámetros formales (*nunca delante de la palabra reservada `template`*).

```
// declaración correcta
template <class T> inline T f(T a, T b)
```

```
{
   // cuerpo de la función
}

// declaración incorrecta
extern template <class T> T f(T a, T b)
{
   // cuerpo de la función
}
```

15.3.3. Un ejemplo de plantilla de funciones

Se desea diseñar una plantilla que calcule el menor valor de dos datos dados, por ejemplo, la función plantilla `min()`, que se define como:

```
template <class T> T min(T a, T b)
{
   if(a <= b)
      return a;
   else
      return b;
}
```

La sintaxis anterior especifica que la función `min()` está parametrizada en la función del *tipo de datos* `T`. Para ver la diferencia entre la función plantilla `min()` y las restantes funciones, obsérvese el programa siguiente:

```
template <class T> min(T a, T b)
{
   if(a <= b)
      return a;
   else
      return b;
}

int main()
{
   int ea = 1; eb = 5;
   cout << "(int): " << min(ea, eb) << endl;

   long 1a = 10000, 1b = 5000;
   cout << "(long): " << min(1a, 1b) << endl;

   char ca = 'a', cb = 'x';
   cout << "(char): " << min(ca, cb) << endl;

   double da = 423.654, db = 789.12;
   cout << "(double): " << min(da, db) << endl;
   return o;
}
```

Cuando el compilador encuentra una llamada de la forma `min(a, b)`, *instancia* la función `min()` a partir de los tipos de parámetros `a` y `b` utilizados en la llamada de la función. Así, el tipo genérico `T` es sustituido por parámetros `ea` y `eb`, `da` y `db`, etc.

Ejemplo 15.1

La plantilla función `min()` se puede declarar también así (archivo `min.h`):

```
template <class T> T min(const T a, const T b)
{
   return a < b ? a : b;
}
```

Algunos ejemplos de llamadas a la plantilla función `min()`:

```
#include "min.h"
int i, j, k;
double u, v, w;
char a, b, c;

...
k = min(i, j);
w = min(u, v);
c = min(a, b);
```

Estas tres llamadas a `min` harán que C++ genere tres funciones plantilla con los siguientes prototipos:

```
int min(const int a, const int b);
char min(const char a, const char b);
double min(const double a, const double b);
```

Ejemplo 15.2

El archivo `minmax.h` declara dos plantillas de funciones: `min()` y `max()`.

```
//Archivo minmax.h

#ifndef _INMAX_H
#define _INMAX_H           //evitar múltiples #include

//plantilla de función max
template <class T> T max(T a, T b)
{
   if(a > b)
      return a;
   else
      return b;
}
// plantilla de función min
template <class T> T min(T a, T b)
{
   if(a < b)
      return a;
   else
      return b;
}

#endif                                // _MINMAX_H
```

Un programa que utiliza funciones plantilla se muestra en el listado `PLANTI.CPP`, donde se incluyen los correspondientes prototipos de funciones.

```
#include <iostream.h>
using namespace std;

#include "minmax.h"

int max(int a, int b);
double max(double a, double b);
char max(char a, char b);

int main()
{
   int e1 = 100, e2 = 200;
   double d1 = 3.141592, d2 = 2.718283;

   cout << "max(e1, e2) es igual a: " << max(e1, e2) << '\n';
   cout << "max(d1, d2) es igual a: " << max(d1, d2) << '\n';
   cout <<-"max(c1, c2) es igual a: " << max(c1, c2) << '\n';
   return 0;
}
```

Ejemplo 15.3

La función plantilla `intercambio` *intercambia dos valores del mismo tipo.*

```
template <class T>
void intercambio(T& a, T& b) {
   T aux = a;
   a = b;
   b = aux;
   return o;
}

int main() {
   int x = 5;
   int y = 12;

   intercambio(x, y)            //ahora x = 12, y = 5
}
```

Así pues, las llamadas

```
intercambio(long1, long2);  //llamada a intercambio (long &, long &);
intercambio(int1, int2);    //llamada a intercambio(int &, int &);
```

15.3.4. Un ejemplo de función plantilla

Se puede utilizar la plantilla `intercambio` para construir una función plantilla `ordenar`:

```
template <class T> void ordenar(T* v, int n)
```

```
{
   for (int intervalo = n/2; intervalo > 0; intervalo /=2)
      for(int i = intervalo; i < n; i++)
         for(int j = i - intervalo;
            j >= 0 && v[j+intervalo] < v[j];j -= intervalo)
               intercambio(v[j], v[j+intervalo]);
}
extern int rango_ent[30];
extern string lista_cadena[10];

ordenar(rango_ent, 30);   //llama a ordenar(int *, int);
ordenar(lista_cadena, 10);//llama a ordenar (string *, int);
```

15.3.5. Plantillas de función `ordenar` y `buscar`

Las plantillas de funciones se utilizan con mucha frecuencia para ordenar listas o buscar elementos en listas. Por ejemplo, se puede definir una función parametrizada ordenar cualquier tipo de array o lista, tal como:

```
template <class T> void ordenar (Array <T>)
{
   // ...
}
```

Este ejemplo declara esencialmente un conjunto de funciones `ordenar` sobrecargadas, una por cada tipo de `Array`. Se puede invocar la función ordenar como si fuera cualquier función ordinaria. El compilador C++ analiza los argumentos de la función y llama a la versión adecuada de la función. Por ejemplo, dadas `eArray` y `fArray` (los arrays de `int` y `float`, respectivamente), se puede aplicar la función `ordenar` para cada una de la forma siguiente:

```
ordenar(eArray);               //ordenar el array de enteros
ordenar(fArray);               //ordenar el array de float
```

Otra función plantilla que implementa una búsqueda genérica en listas puede tener el siguiente código:

```
template <class T>
unsigned BusquedaBin(T& DatosABuscar, T lista[], unsigned num)
{
   in primero = 0, último = num –1;
   unsigned m;

   do {
      m = (primero + último)/2;
      if (DatosABuscar < lista[m])
         ultimo = m–1;
      else
         primero = m + 1;
   } while (!DatosABuscar == array[m] ||primero > ultimo);

   //
```

```
   return (DatosABuscar == array[m]? m: 0xffff;
}
```

15.3.6. Una aplicación práctica

Diseñar una plantilla de función para calcular el máximo de dos datos y un programa que haga uso de esa plantilla.

```
template <class Tipo>
Tipo max2(Tipo primero, Tipo segundo)
{
   return primero > segundo? primero: segundo;
}

template <class Tipo>
Tipo max3(Tipo primero, Tipo segundo, Tipo tercero)
{
   return max2(max2 (primero, segundo), tercero);
}
```

Un código que utiliza esta función plantilla es:

```
void main()
{
   cout << max2(6, 4) << "\n";
   cout << max3(10, 40, 20) << "\n";
   cout << max3(9.99, 4.45, 3.1416) << "\n";
   cout << max3('M', 'A', 'S',) << "\n";
}
```

Al ejecutarse el programa anterior se visualizará:

```
6
40
9.99
S
```

Precaución

Suponga que se realizan dos invocaciones:

```
cout << max2(4, 5.9) << "\n";
cout << max2('A', 66) << "\n";
```

Al compilar este segmento de programa se producirán dos errores en tiempo de compilación. Una solución posible para evitar estos errores de compilación es definir una nueva función plantilla, como la siguiente:

```
template <class Tipo1, class Tipo2>
Tipo1 maxd(Tipo1 primero, Tipo2 segundo)
```

```
{
   return primero > (Tipo1) segundo ? primero : (Tipo1) segundo;
}
```

El resultado será del tipo del primer parámetro, y el siguiente código:

```
void main()
{
   cout << maxd(4, 5.9) << "\n";
   cout << maxd('A', 66) << "\n";
}
```

producirá los resultados siguientes:

```
5
B
```

15.3.7. Problemas en las funciones plantilla

Cuando se declaran plantillas de funciones con más de un parámetro, para evitar errores es preciso tener mucha precaución.

Así, por ejemplo, si la función se declara con múltiples parámetros y devuelve un valor de tipo `T`:

```
template <class T> T f(int a, T b)
{
   //cuerpo de la función
}
```

Esta versión de la función plantilla devuelve un tipo `T` y tiene dos parámetros: un parámetro `int` llamado `a` y un tipo objeto no especificado `T` llamado `b`. El usuario de la función plantilla ha de proporcionar el tipo de dato para `T`. Por ejemplo, un programa puede especificar el prototipo:

```
double f(int a, double b);
```

Consideremos otro ejemplo, la función `min` definida anteriormente:

```
// archivo MINAUX.CPP
template <class T> min (T a, T b)
{
   if (a < b)
      return a;
   else
      return b;
}

int main()
{
   char c1 = 'J', c2 = '1';
   int n1 = 25, n2 = 65;
   long n3 = 50000;
   float n4 = 84.25, n5 = 9.999;
```

```
   min(c1, c2);           // correcto
   min(n1, n2);           // correcto
   min(n4, n5);           // correcto
   min(c1, n1);           // error
   min(n3, n4);           // error
   min(n2, n3);           // error
}
```

La compilación de este programa genera errores debido a la discordancia de tipos en las siguientes líneas:

```
min(c1, n1);              // c1 y n1 tipos distintos
min(n3, n4);              // n3 y n4 tipos distintos
min(n2, n3);              // n2 y n3 tipos distintos
```

15.4. PLANTILLAS DE CLASES

Las *plantillas de clase* permiten definir clases genéricas que pueden manipular diferentes tipos de datos. Una aplicación importante es la implementación de *contenedores*, clases que contienen objetos de un tipo dato, tales como vectores (*arrays*), listas, secuencias ordenadas, tablas de dispersión (*hash*); en esencia, los contenedores manejan estructuras de datos.

Así, es posible utilizar una clase plantilla para crear una pila genérica, por ejemplo, que se puede instanciar para diversos tipos de datos predefinidos y definidos por el usuario. Puede tener también clases plantillas para colas, vectores (*arrays*), matrices, listas, árboles, tablas (*hash*), grafos y cualquier otra estructura de datos de propósito general. En terminología orientada a objetos, las plantillas de clases se denominan también *clases parametrizables*.

15.4.1. Definición de una plantilla de clase

La sintaxis de la plantilla de clase es:

```
template <class nombretipo> class tipop {
   //...
};
```

donde *nombretipo* es el nombre del tipo definido por el usuario utilizado por la plantilla —tipo genérico T— y *tipop* es el nombre del tipo parametrizado para la plantilla (es decir, tipop es su *clase genérica)*. T no está limitado a clases o tipos de datos definidos por el usuario y puede tomar incluso el valor de tipos de datos aritméticos (int, char, float, etc.).

Al igual que en las plantillas de funciones, el código de la plantilla siempre está precedido por una sentencia en la cual se declara T como parámetro de tipo (pueden definirse tambien varios parámetros tipo en las plantillas de clases).

La sintaxis básica de una plantilla de clase es una cabecera de la declaración de la clase seguida por una declaración o definición de clase.

Sintaxis

```
template <typename T>
class Pila {
   ...
};
```

alternativamente

```
template <class T>
class Pila {
   ...
};
```

Ejemplo

```
template <typename T>
struct Punto {
   T x, y;
};
```

Para utilizar la plantilla de clase, proporcione un argumento para cada parámetro de la plantilla:

```
punto<int> pt = {45, 15};
```

De acuerdo a la sintaxis propuesta, la plantilla para una clase genérica `Pila` se puede escribir así:

```
// archivo PILAGEN.H
// interfaz de una plantilla de clases para definir pilas

template <class T>
class Pila
{
   T datos[50]
   int elementos;
   public ;
   Pila(): elementos(0)  {}
   // añadir un elemento a la pila
   void Meter(T elem);
   // obtener un elemento de la pila
   T sacar();
   // número de elementos reales en la pila
   int Numero();
   // ¿está la pila vacía?
   bool vacia();
};
```

El prefijo `template <class T>` en la declaración de clases indica que se declara una plantilla de clase y que se utilizará `T` como el tipo genérico. Por consiguiente, `Pila` es una clase parametrizada con el tipo `T` como parámetro.

Con esta definición de la plantilla de clases `Pila` se pueden crear pilas de diferentes tipos de datos, tales como:

```
Pila <int> pila_ent;      // Una pila para variables int
Pila <float> pila_real;   // Una pila para variables float
```

De igual modo, se define una clase genérica `Array` como sigue:

```
template <class T> class Array
{
```

```
public :
   Array(int n = 16) {pa = new T[_longitud = n];}
   ˜Array() {delete[] pa;}
   T& operator[] (int i);
// ...
private :
   f* pa;
   int longitud;
};
```

Se puede crear, a continuación, diferentes tipos de arrays de la siguiente forma:

```
Array <int> intArray(128);     // Array de 128 elementos int
Array <float> fArray(32);      // Array de 32 elementos float
```

Consideremos la siguiente especificación de la clase plantilla `cola`, que contiene dos parámetros:

```
template <class elem, int tamaño> class cola {
   int tamaño;
   ...
public;
   cola(int n);
   int vacia();
   ...
};
```

La clase plantilla `cola` tiene dos parámetros plantilla: una variable de tipo `elem`, que especifica el tipo elemento cola, y `tamaño`, que especifica el tamaño de la cola.

Algunas definiciones de variables que ilustran el uso de la clase plantilla `cola`:

```
cola <int, 2048> a;
cola <char, 512> b;
cola <char, 1024> c;
cola <char, 512*2> d;
```

Dos nombres de clase plantillas se refieren a las mismas clases, sólo si los nombres de plantillas son idénticos y sus argumentos tienen valores idénticos. En consecuencia, sólo las variables `c` y `d` tienen los mismos tipos.

La implementación de una clase plantilla requerirá unas funciones constructor, destructor y miembros. Así, una definición de un constructor de plantilla tiene el formato:

```
template <declaraciones-parámetro-plantilla>
         nombre-clase <parámetros-plantilla>::nombre_clase
{
   // ...
}
```

El cuerpo del constructor de la plantilla `cola`:

```
template <class elem, int tamaño>
   cola <elem, tamaño>::cola (int n)
{
   ...
}
```

Las definiciones de destructores son similares a las definiciones de los constructores.

Una definición de una función miembro `vacía` de una plantilla de la clase `cola` tiene el formato siguiente:

```
template <declaraciones-parámeteros-plantilla> tipo-resultado
         nombre-clase <parámetros-plantilla>::
         nombre-func-miembro(declaraciones-parámetros)
{
   // ...
}
```

Como ejemplo de la sintaxis anterior se puede definir la función `vacía` de la clase plantilla `cola`:

```
template <class elem, int tam> int cola <elem, tam>::vacía()
{
   // ...
}
```

15.4.2. Instanciación de una plantilla de clases

Al igual que con las plantillas de funciones, se pueden instanciar las plantillas de clases. Una *clase plantilla* es una clase construida a partir de una plantilla de clases. La plantilla de clases ha de ser instanciada para manipular los objetos del tipo adecuado. Es decir, cuando el compilador se encuentra especificado de tipo plantilla, tal como:

```
cola <int, 2048> a;
```

la primera vez toma los argumentos dados para la plantilla y construye una definición de clase automáticamente (Fig. 15.2).

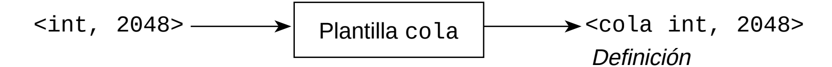


Figura 15.2. Instanciación de una plantilla.

15.4.3. Utilización de una plantilla de clase

La manipulación de una plantilla de clase requiere tres etapas:

- Declaración del tipo parametrizado (por ejemplo `Pila`).
- Implementación de la pila.
- Creación de una instancia específica de pila (por ejemplo, datos de tipo entero `-int-` o carácter `char`).

Así, por ejemplo, supongamos que se desea crear un tipo parametrizado `Pila` con las funciones miembro `poner` y `quitar`.

Declaración de la plantilla `Pila`

```
template <class Tipo>
class Pila {
```

```
public:
   Pila();
   bool poner(const Tipo);   // meter elemento en la pila
   bool quitar(Tipo&);       // sacar elemento de la pila
private :
   Tipo elementos[MaxElementos];// elementos de pila
   int cima;                    // cima de la pila
};
```

Implementación de la pila

```
template <class Tipo>
Pila <Tipo>:: Pila()
{
   cima = -1;
}

template <class Tipo>          // función miembro poner
bool Pila <Tipo>:: poner(const Tipo item)
{
   if(cima < MaxElementos -1)  {
      elementos[++cima] = item;
      return true;
   }
   else {
      return false;
   }
}

template <class Tipo>
bool Pila <Tipo> :: quitar(Tipo& item)
{
   if (cima  < o) {
      return false;
   }
   else {
      item = elementos[cima--];
      return true;
   }
}
```

Instanciación de la plantilla de clases

Una instancia de una pila específica se puede instanciar mediante:

```
Pila <int> pila_ent;          // pila de enteros
Pila <char> pila_car;         // pila de caracteres
```

> ***Nota***
>
> Mediante un tipo parametrizado no se puede utilizar una pila que se componga de tipos diferentes. Posteriormente, se verá qué objetos polimórficos podrían realizar el efecto de tener tipos diferentes de objetos procesados a la vez, aunque no siempre esto es lo que se requiere.

15.4.4. Argumentos de plantillas

Los argumentos de plantilla no se restringen a tipos, aunque éste sea uno predominante. Los parámetros de una plantilla pueden ser cadenas de caracteres, nombres de funciones y expresiones de constantes.

Un caso interesante es el uso de una constante entera para definir el «tamaño» de una estructura de datos de tipo genérico. Por ejemplo, el siguiente código declara un vector genérico de `n` elementos:

```
template <class T, int n>
class vector
{
   T datos[n];
   // ...
};
```

Este argumento constante puede incluso tener un valor por defecto, tal como argumentos normales de funciones. La regla de compatibilidad de tipos entre instancias de argumentos de plantilla permanece igual: dos instancias son compatibles si sus argumentos tipo son iguales y sus argumentos expresiones tienen el mismo valor. Esta regla significa que las declaraciones siguientes definen dos objetos compatibles:

```
Vector <float, 100> V1,
Vector <float, 25*4> V2;
```

15.4.5. Aplicación de plantillas de clases

Diseño de la plantilla de clase `Pila`*, con separación de archivos de cabecera* (`PILAGEN2.h`) *y archivo de implementación* (`PILAGEN2.CPP`).

Estas clases de contenedores genéricos son muy útiles. Por ejemplo, se puede reescribir la clase plantilla `Pila` sin tener que especificar un número fijo de elementos. También se puede añadir fácilmente una función para comprobar si la pila está llena. La interfaz de la clase es el siguiente:

```
// archivo PILAS.H
// interfaz de una plantilla de clases para definir pilas
// con el número de elementos definidos en la instanciación

template <class T, int nE1 = 100> class Pila
{
   T datos[nE1];
   int nElementos;
public:
   Pila():nElementos(0){}
   // añade un elemento a la pila
   void Poner (T elem);
   // obtener un elemento de la pila
   T Quitar();
   // número de elementos reales en la pila
   int Numero();
   //¿está la pila vacía?
   bool Vacía();
```

```
   // ¿está la pila llena?
   bool Llena();
};
```

La definición de las funciones miembro utiliza una sintaxis tal vez más compleja para especificar los dos parámetros de plantilla, como se muestra a continuación:

```
// archivo PILAS.CPP
// definición de funciones miembro de una plantilla de clase
// Pila con un número indefinido de elementos.
#include "Pilas.h"

template <class T, int nE1>
void Pila <T, nE1>::Poner(T elem)
{
   datos[nElementos] = elem;
   nElementos++;
}

template <class T, int nE1>
T Pila <T, nE1> ::Quitar()
{
   nElementos--;
   return datos[nElementos];
}

template <class T  int nE1>
int Pila <T, nE1>::Numero()
{
   return nElementos;
}

template <class T, int nE1>
int Pila <T, nE1> ::Vacia()
{
   return(nElementos == 0);
}

template <class T, int nE1>
int Pila <T, nE1> ::Llena()
{
   return(nElementos == nE1;
}
```

Una propiedad interesante de esta clase es que el segundo parámetro se puede omitir, utilizando el valor por defecto en el siguiente ejemplo:

```
// archivo USAPILAS.CPP
// ejemplo de una instancia de una plantilla de clase Pila con un
// número genérico de elementos
```

```
#include "Pilas.cpp"
void main()
{
   // definición de una pila de números enteros (tamaño por defecto)
   Pila <int> PilaEnt;
   // definir una pila pequeña de datos double
   Pila <double, 5> miniPila;
}
```

15.5. UNA PLANTILLA PARA MANEJO DE PILAS DE DATOS[5]

Se trata de diseñar una clase `Pila` que permita manipular pilas de diferentes tipos de datos. Una *pila* es una estructura de datos que permite almacenar datos de modo que el último dato en *entrar* en la pila es el primero en *salir*.

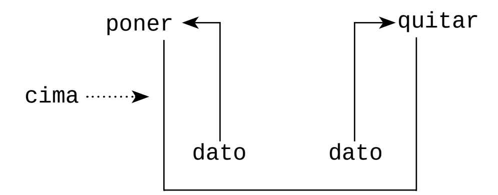


Figura 15.3. Estructura de datos *pila*.

Las operaciones que se consideran son `poner`, `quitar` y `visualizar`.

```
// archivo PILA1.CPP

enum estado_pila {OK, LLENA, VACIA};
template <class T> class Pila
{
public:
   Pila(int _longitud = 10);
   ˜Pila()  {delete[] tabla;}
   void poner(T);
   T quitar();
   void visualizar();
   int num_elementos();
   int leer_long()  {return longitud;}
private:
   int longitud;
   T* tabla;
   int cima;
   estado_pila estado;
};
```

En la declaración anterior se considera el tipo genérico `T`. `Pila` es una clase parametrizada por un tipo `T` y se representa con la notación `Pila <T>`.

[5] En el Capítulo 18 se ampliará y estudiará en profundidad el tipo de estructura de datos, *Pila*.

15.5.1. Definición de las funciones miembro

La declaración de cada función se precede por `template <class T>` que indica al compilador que la función está parametrizada por el tipo `T`.

```
template <class T> Pila <T>::Pila(int _longitud)
{
   longitud = _longitud;
   tabla    = new T[longitud];
   cima     = 0;
   estado   = VACIA;
}

template <class T> void Pila <T>::poner(T _elemento)
   if (estado!= LLENA)
   tabla [++cima] = _elemento;
   else
      cout << "*** Pila llena ***" << endl;
   if (cima >= longitud)
      estado   = LLENA;
   else
      estado = OK;
}

template <class T> T Pila <T> :: quitar()
{
   T elemento = 0;

   if (estado!= VACIA)
      elemento = tabla[--cima];
   else
   cout << "*** Pila vacía ***" << endl;

   if (cima <= 0)
      estado = VACIA;
   else
      estado = OK;

   return elemento;
}

template <class T> void Pila <T>::visualizar()
{
   for (int i = cima; i > 0, i--)
      cout <<"[" << tabla[i] << "]  << endl;
}

template <class T> int Pila<T>:: num_elementos()
{
   return cima;
}
```

15.5.2. Utilización de una clase plantilla

La manipulación de la pila con diversos tipos de datos se puede comprobar con el siguiente programa:

```
void main()
{
   // Pila de enteros

   Pila <int> p1(6);

   p1.poner(6);
   p1.poner(12);
   p1.poner(18);

   cout << "Número de elementos:" << p1.num_elementos() << endl;
   p1.visualizar();

   cout <<"Quitar 1 :" << p1.quitar() << endl;
   cout <<"Quitar 2 :" << p1.quitar() << endl;
   cout <<"Quitar 3 :" << p1.quitar() << endl;
   cout <<"Número de elementos :" <<p1.num_elementos() << endl;
   cout <<"Quitar 4 :" << p1.quitar() << endl;

   cout << endl ;

   // Pila de enteros largos
   Pila <long> p2(6);
   p2.poner(60000L);
   p2.poner(1000000L);
   p2.poner(2000000L);

   cout <<"Número de elementos:" << p2.num_elementos() << endl;
   p2.visualizar();

   cout <<"Quitar 1 : " << p2.quitar() << endl;
   cout <<"Quitar 2 : " << p2.quitar() << endl;
   cout <<"Quitar 3 : " << p2.quitar() << endl;
   cout <<"Número de elementos :" << p2.num_elemento << endl;
   cout <<"Quitar 4 : " << p2.quitar() << endl;

   Pila <double> p3(6);

   p3.poner(6.6);
   p3.poner(12.12);
   p3.poner(18.18);

   cout <<"Número de elementos : " << p3.num_elementos() << endl;
   p3.visualizar();

   cout << "Quitar 1 : " <<p3.quitar() << endl;
   cout << "Quitar 2 : " <<p3.quitar() << endl;
   cout << "Quitar 3 : " <<p3.quitar() << endl;
```

```
cout << "Número de elementos :" << p3.num_elementos()<< endl;
cout << "Quitar 4 : " <<p3.quitar() << endl;
}
```

La ejecución de este programa proporciona la siguiente salida:

```
Número de elementos : 3
[18]
[12]
[6]
Quitar 1 : 18
Quitar 2 : 12
Quitar-3 : 6
Números de elementos : 0
*** Pila vacía ***
Quitar 4 : 0

Número de elementos : 3
[2000000]
[1000000]
[60000]
Quitar 1 : 2000000
Quitar 2 : 1000000
Quitar 3 : 60000
Número de elementos : 0
*** Pila vacía ***
Quitar 4 : 0

Número de elementos : 3
[18.18]
[12.12]
[6.6]
Quitar 1 : 18.18
Quitar 2 : 12.12
Quitar 3 : 6.6
Número de elementos : 0
*** Pila vacía ***
Quitar 4 : 0
```

Otro método de implementación de la estructura pila se muestra en el archivo `PilaGen.h`.

```
// FICHERO. PilaGen.h

#ifndef PILAGEN4_H
#define PILAGEN4_H

template <class Tipo>
class Pila
{
public :
   Pila();
   Pila(const unsigned n);
```

```
   Pila();
   void Vaciar();
   void Poner(const Tipo & x);
   Tipo Quitar();
   Tipo Cima() const;
   bool Vacía() const;
   bool Llena() const;
private:
   unsigned max ;
   unsigned cima;
   Tipo * valor;
};

template <class Tipo>
Pila<Tipo>::Pila():
   max(100),
   cima(0),
   valor(new Tipo[max])
{}

template <class Tipo>
Pila<Tipo>::Pila(const unsigned n):
   max(n),
   cima(0),
   valor(new Tipo[max]);
{}

template <class Tipo>
Pila<Tipo>::˜Pila()
{
   delete []valor;
}

template <class Tipo>
void Pila<Tipo>::Vaciar()
{
   cima = 0;
}

template <class Tipo>
void Pila<Tipo>::Poner(const Tipo & x)
{
   valor[cima++] = x;
}

template <class Tipo>
Tipo Pila<Tipo>::Quitar()
{
   return valor[--cima];
}

template <class Tipo>
```

```
Tipo Pila<Tipo>::Cima() const
{
   return cima == 0;
}

template <class Tipo>
bool Pila<Tipo>::Llena() const
{
   return (cima >= max) ;
}

#endif // PILAGEN4_H
```

15.5.3. Instanciación de una clase plantilla con clases

Una clase plantilla se puede instanciar con cualquier tipo de dato; por ejemplo, en la aplicación anterior `Pila` con objetos de *tipo Complejo* o *tipo Cadena*. Así, por ejemplo, suponiendo que se dispone de dos clases, `Complejo` y `Cadena`, que permiten realizar operaciones sobre números complejos y cadenas de caracteres (*strings*). Se instancia fácilmente la plantilla `Pila` para manipular los tipos de datos citados.

```
void main()
{
   // Pila de complejos
   Pila <Complejo> p1(5);

   p1.poner(Complejo(5, -4));
   p1.poner(Complejo(10, -2));
   p1.poner(Complejo(15, -3));

   cout << "Número de elementos : " << p1.num_elementos()<< endl;
   p1.visualizar();

   // Pila de Cadenas

   Pila <Cadena> p2(5);

   p2.poner("Prueba primera");
   p2.poner("Prueba segunda");
   p2.poner("Prueba tercera");

   cout << " Número de elementos : " << ps.num_elementos() << endl;
   p2.visualizar();
}
```

15.5.4. Uso de las plantillas de funciones con clases

Las funciones plantilla se pueden utilizar con clases. Como es normal, la clase definirá la operación realizada sobre el objeto en la función plantilla. En el programa siguiente, la función plantilla `min` opera sobre los elementos de una clase numérica.

```
// archivo PLANTI.CPP
// ejemplo de función plantilla utilizada con una clase

template <class T> T min(Ta, Tb)
{
   if (a < b)
      return a;
   else
      return b;
}
class Numero
{
   long num;
   public:
   Numero(long n):num(n)
   {}
   long Valor()
   {
      return num;
   }

   int operator < (Numero n2)
   {
      return num < n2.Valor();
   }
};

void main()
{
   Numero nn1 = 15;
   Numero nn2 = 25;
   Numero nn3 = min(nn1, nn2);
}
```

15.6. MODELOS DE COMPILACIÓN DE PLANTILLAS[5]

Cuando el compilador ve una definición de plantilla, no se genera código inmediatamente. El compilador produce instancias específicas de tipos de la plantilla sólo cuando vea una llamada de la plantilla, tal como cuando se llama una plantilla de función se llama o un objeto de una plantilla de clase se define.

Normalmente, cuando se invoca a una función, el compilador necesita ver sólo una declaración de esa función. De modo similar, cuando se define un objeto de un tipo clase, la definición de la clase debe estar disponible, pero las definiciones de las funciones miembro no necesitan estar presentes. Como resultado se ponen las declaraciones de la función y las definiciones de la clase en archivos cabecera y las definiciones de funciones ordinarias y miembros de la clase en archivos fuente.

Las plantillas son diferentes [Lippman, 2005]. Para generar una *instantación* el compilador debe terner que acceder al código fuente que define la plantilla. Cuando se llama a una plantilla de función o una función miembro de una plantilla de clase, el compilador necesita la definición de la función; se necesita el código fuente que está en los archivos fuente.

[5] Lippman, Lajoie y Moo realizan una excelente explicación de programación genérica y plantillas en [Lippman, 2005] que recomendamos al lector. [Lippman, 2005]. *C++ Primer*, 4.ª ed. Addign Werley, 2005.

C++ estándar define dos modelos para compilación del código de las plantillas [Lippman, 2005]. Ambos modelos estructuran los programas de un modo similar: las definiciones de las clases y las declaraciones de las funciones van en archivos de cabeza y las definiciones de miembros y funciones van en archivos fuente. Los dos modelos difieren en el modo que las definiciones de los archivos fuentes se ponen disponibles al compilador. Todos los compiladores soportan el modelo denominado «inclusión» y sólo algunos compiladores soportan el modo de «compilación reparada».

15.6.1. Modelo de compilación de inclusión

En el modelo de inclusión, el compilador debe ver la definición de cualquier plantilla que se utilice. La solución que se adopta es incluir en el archivo de cabecera no sólo las declaraciones, sino también las definiciones. Esta estrategia permite mantener la separación de los archivos de cabecera y los archivos de implementación, aunque se incluye una directiva `#include` en el archivo de cabecera para que inserte las definiciones del archivo `.ccp`.

Ejemplo

Escribir los archivos necesarios para implementar una clase. Normalmente, las definiciones se hacen disponibles añadiendo una directiva `#include` a las cabeceras que declaran las plantillas de clases o de función. Esta `#include` lleva los archivos fuente que contienen las definiciones variables.

```
//archivo de cabecera demo.h
#ifndef DEMO_H
#define DEMO_H
template<class T>
int comparar(const T&, const T&);
//otras declaraciones

#include "demo.cc"   //definiciones de comparar
#endif

//implementación del archivo demo.cc
template<class T> int comparar(const T &a, const T &b)
{
   if(a < b) return -1;
   if(b < a) return 1;
   return 0;
}
//otras definiciones
```

Este método permite mantener la separación de los archivos de cabecera y los archivos de implementación pero asegura que el compilador verá ambos archivos cuando se compila el código que utiliza la plantilla.

Algunos compiladores que utilizan el modelo de inclusión pueden generar instanciaciones múltiples. Si dos o más archivos fuente compilados por separado utilizan la misma plantilla, estos compiladores generan una instanciación para la plantilla en cada archivo. Esto supondrá que una plantilla se puede instanciar más de una vez.

15.6.2. Modelo de compilación separada

Este modelo de compilación es muy parecido al modelo típico de C++ y permite escribir las declaraciones y funciones en dos archivos (extensiones `.h` y `.cpp`). La única condición es que se debe utilizar la

palabra reservada `export` para conseguir la compilación separada de definiciones de plantillas y declaraciones de funciones de plantillas. Sin embargo, este modelo de compilación no está implementada en la mayoría de los compiladores.

La palabra reservada `export` indica que una definición dada puede ser necesaria para generar instanciaciones en otros archivos. Una plantilla puede ser definida como exportada solamente una vez en un programa. El compilador decide cómo localizar la definición de la plantilla cuando se necesiten generar estas implantaciones. La palabra reservada `export` no necesita aparecer en la declaración de la plantilla. Normalmente, se indica que una plantilla de funciones sea *exportada* como parte de su definición. Se debe incluir la palabra reservada `export` antes de la palabra reservada `template`.

Ejemplo

La declaración de la plantilla de función se pone en un archivo de cabecera, pero la declaración no debe especificar `export`.

```
//definición de la plantilla en un archivo compilado por separado
export template<typename T>
T suma(T t1, T t2)
```

El uso de `export` en una plantilla de clase es un poco más complicado. Como es normal, la declaración de la clase debe ir en un archivo de cabecera. El cuerpo de la clase en la cabecera no utiliza la palabra reservada `export`. Si se utiliza `export` en la cabecera, entonces esa cabecera se utilizará en un único archivo fuente del programa.

```
//cabecera de la plantilla de clase está en el archivo
//de cabecera compartido
template <class T> class Pila {...};
//Archivo pila.ccp declara Pila como exportada
export template <class T> class Pila;
#include "Pila.h"
//definiciones de funciones miembro de Pila
```

Los miembros de una clase exportada se declaran automáticamente como exportados; también se pueden declarar miembros individuales de una plantilla de clase como exportados. En este caso, la palabra `export` no se especifica en la plantilla de clase; sólo se especifica en las definiciones de los miembros específicos que se exportan. La definición se debe situar dentro del archivo de cabecera que define la plantilla de clase.

> **Nota**
>
> La compilación separada es muy interesante pero no es fácil su implementación. Por otra parte, esta característica sólo está implementada en las últimas generaciones de compiladores y pudiera suceder que su propio compilador no la incorpore.

15.7. PLANTILLAS FRENTE A POLIMORFISMO

Una pregunta usual tras estudiar las plantillas y el polimorfismo es conocer cuáles son las diferencias entre estos conceptos. ¿Se puede sustituir el polimorfismo por una plantilla? ¿Cuál de las dos características es mejor? Para dar una contestación repasemos ambos conceptos.

Una función es polimórfica si uno de sus parámetros puede suponer tipos de datos diferentes, que deben derivarse de los parámetros formales. Cualquier función que tenga un parámetro como puntero a una clase puede ser una función polimórfica y se puede utilizar con tipos de datos diferentes.

Una función es una función plantilla sólo si está precedida por una cláusula `template` apropiada. Esta definición significa que estas funciones se diseñan, definen e implementan pensando en genericidad. Por consiguiente, escribir una función plantilla implica pensar en genericidad, evitando cualquier dependencia en tipos de datos, constantes numéricas, etc.

Una función plantilla es sólo una plantilla (un patrón) y no una verdadera función. Aunque no necesita ser instanciada, este proceso se hace automáticamente por el compilador para cualquier llamada diferente. Como resultado se dispone de una familia de funciones sobrecargadas, que tienen todas el mismo nombre y parámetros diferentes. En otras palabras, la cláusula plantilla es un generador automático de funciones sobrecargadas.

Las funciones polimórficas son funciones que se pueden ejecutar dinámicamente con parámetros de tipos diferentes. Por otra parte, las funciones plantilla son funciones que se pueden compilar estáticamente en versiones diferentes para acomodarse con parámetros de tipos de datos diferentes.

Desde el punto de vista de usos prácticos de estas dos construcciones, se puede observar que:

- Las funciones plantillas trabajan también con tipos aritméticos.
- Las funciones polimórficas deben utilizar punteros.
- La genericidad polimórfica se limita a jerarquías.
- Las plantillas tienden a generar un código ejecutable grande, dado que se duplican las funciones.

RESUMEN

En el mundo real, cuando se define una clase o función, se puede desear poder utilizarla con objetos de tipos diferentes, sin tener que reescribir el código varias veces. Las últimas versiones de C++ incorporan las plantillas (*templates*) que permiten declarar una clase sin especificar el tipo de uno o más miembros datos (esta operación se puede retardar hasta que un objeto de esa clase se define realmente). De modo similar, se puede definir una función sin especificar el tipo de uno o más parámetros hasta que la función se llama.

Para declarar una familia completa de clases o funciones se puede utilizar la cláusula `template`. Con plantillas sólo se necesita seleccionar la clase característica de la familia.

Las plantillas proporcionan la implementación de tipos parametrizados o genéricos. La *genericidad* es una construcción muy importante en lenguajes de programación orientados a objetos. Una definición muy acertada se debe a Meyer:

> «Genericidad es la capacidad de definir módulos parametrizados. Tal módulo se denomina módulo genérico, no es útil directamente; en su lugar, es un patrón de módulos. En la mayoría de los casos, los parámetros representan tipos. Los módulos reales denominados instancias del módulo genérico se obtienen proporcionando tipos reales para cada uno de los parámetros genéricos» [Meyer, 88].

El propósito de la genericidad es definir una clase (o una función) sin especificar el tipo de uno o más de sus miembros (parámetros).

Las plantillas permiten especificar un rango de funciones relacionadas (sobrecargados), denominadas *funciones plantilla*, o un rango de clases relacionadas, denominadas *clases plantilla*.

Todas las definiciones de plantillas de funciones comienzan con la palabra reservada `template`, seguida por una lista de parámetros formales encerrados entre ángulos (< >); cada parámetro formal debe estar precedido por la palabra reservada `class`. Algunos patrones de sintaxis son:

1. `template<class T>`
2. `template<class TipoElemento>`
3. `template<class TipoBorde, class TipoRelleno>`

Las plantillas de clases proporcionan el medio para describir una clase genéricamente y se instancian con versiones de esta clase genérica de tipo específico. Una plantilla típica tiene el siguiente formato:

```
template<class T>
class Demo
{
   T v;
   ...
public:
   Demo (const T & val):v(val) {}
   ...
};
```

Las definiciones de las clases (*instanciaciones*) se generan cuando se declara un objeto de la clase especificando un tipo específico. Por ejemplo, la declaración

```
Demo<short> ic;
```

hace que el compilador genere una declaración de la clase en el que cada ocurrencia del tipo de parámetro T en la plantilla se reemplaza por el tipo real `short` en la declaración de la clase. En este caso, el nombre de la clase es `Demo<short>` y no `Demo`.

El objetivo de la genericidad es permitir reutilizar el código comprobado sin tener que copiarlo manualmente. Esta propiedad simplifica la tarea de programación y hace los programas más fiables.

- C++ proporciona las plantillas (`templates`) para proporcionar *genericidad* y polimorfismo paramétrico. El mismo código se utiliza con diferentes tipos, donde el tipo es un parámetro del cuerpo del código.
- Una plantilla de función es un mecanismo para generar una nueva función.
- Una plantilla de clases es un mecanismo para la generación de una nueva clase.
- Un parámetro de una plantilla puede ser o bien un tipo o un valor.
- El tipo de los parámetros de la plantilla es una definición de una plantilla de función, se puede utilizar para especificar el tipo de retorno y los tipos de parámetros de la función generada.
- C++ estándar describe una biblioteca estándar de plantilla que, en parte, incluye versiones de plantillas sobre tareas típicas de computación tales como búsqueda y ordenación.

EJERCICIOS

15.1. Definir plantillas de funciones `min()` y `max()` que calcule el valor mínimo y máximo de dos valores.

15.2. Realizar un programa que utilice las funciones plantilla del ejercicio anterior para calcular los valores máximos de parejas de enteros, de doble precisión (`double`) y de carácter (`char`).

15.3. Escribir una clase plantilla que pueda almacenar una pequeña base de datos de registros.

15.4. Realizar un programa que utilice la plantilla del ejercicio anterior para crear un objeto de la clase de base de datos.

15.5. Escribir una plantilla de función `tintercambio` que intercambia dos variables de cualquier tipo.

15.6. Definir una clase plantilla para pilas.

15.7. ¿Cómo se implementan funciones genéricas en C++? Compárela con plantillas de funciones.

15.8. Declarar una plantilla para la función `gsort` para ordenar arrays de un tipo dado.

15.9. Definir una función plantilla que devuelva el valor absoluto de cualquier tipo de dato incorporado o predefinido pasado a ella.

15.10. Definir una función plantilla `max()` que trabaje con tipos de datos predefinidos.

P A R T E III

ESTRUCTURAS DE DATOS

CAPÍTULO 16

Flujos y archivos: biblioteca estándar E/S

Contenido

INTRODUCCIÓN

Hasta este momento se han realizado las operaciones básicas de entrada y salida. La operación de introducir (*leer*) datos en el sistema se denomina **lectura** y la generación de datos del sistema se denomina **escritura**. La lectura de datos se realiza desde su teclado e incluso desde su unidad de disco, y la escritura de datos se realiza en el monitor y en la impresora de su sistema.

Al igual que sucede en C ANSI, las funciones de entrada/salida no están definidas en el propio lenguaje C++, sino que están incorporadas en cada compilador de C++ bajo la forma de *biblioteca de ejecución*. En C existe la biblioteca *stdio.h* estandarizada por ANSI; en C++ su biblioteca correspondiente es *iostream*, aunque en este caso todavía no está estandarizada. En consecuencia, el lector puede recurrir en las operaciones de E/S (Entrada/Salida) a cualquiera de las dos bibliotecas, aunque la biblioteca *iostream* se distingue positivamente de *stdio.h* en que la entrada/salida se realiza por **flujos** (*streams*), método que además de proporcionar la funcionalidad de C ofrece un método flexible y eficiente mediante la gestión de clases, que permite sobrecargar funciones y operadores, lo que hará que sus clases puedan ser manipuladas como si fueran tipos predefinidos.

La gran ventaja de la biblioteca *iostream* es que se pueden manipular las operaciones E/S sin necesidad de conocer los conceptos típicos de orientación a objetos, tales como clases, herencia, funciones virtuales, etc. Una descripción de estos conceptos se proporcionarán en la última parte del libro, a título de introducción en las técnicas orientadas a objetos, aunque no es el objeto principal del libro.

En este capítulo aprenderá a utilizar las características típicas de E/S de C++ y obtener el máximo rendimiento de las mismas.

La biblioteca de flujos `iostream` de C++ contiene la clase **ios**. Esta clase declara los identificadores que establecen el modo de flujos de archivos. La biblioteca tiene las clases `ifstream`, `ofstream` y `fstream`, que soportan flujos de archivos de entrada, de salida y de entrada/salida. Este capítulo examina las funciones de la biblioteca de flujos que permite soportar E/S de archivos de texto, E/S de archivos binarios y E/S de archivos de acceso aleatorio.

Los tipos de clases proporcionan el siguiente soporte de archivos:

- `ifstream`, derivada de `istream`, conecta un archivo al programa para entrada.
- `ofstream`, derivada de `ostream`, conecta un archivo al programa para salida.
- `fstream`, derivada de `iostream`, conecta un archivo al programa para entrada y salida.

Para usar el componente de flujos de archivos de la biblioteca `iostream` se debe incluir su archivo asociado: `#include <fstream>`.

CONCEPTOS CLAVE

- Apertura del archivo.
- Archivo binario.
- Archivo de acceso aleatorio.
- Archivo de cabecera.
- Archivo de texto.
- Biblioteca.
- Biblioteca de clases.
- Bit de estado.
- Cierre del archivo.
- Entrada/Salida.
- Final del archivo (`eof`).
- Flujo (*stream*).
- Flujo binario.
- Flujo de texto.
- Indicador de estado.
- Manipulador.
- Operador de extración.
- Operador de inserción.

16.1. FLUJOS *(STREAMS)*

Un **flujo** (*stream*) es una abstracción que se refiere a un *flujo* o *corriente* de datos que fluyen entre un origen o fuente (*productor*) y un destino o sumidero (*consumidor*). Entre el origen y el destino debe existir una conexión o tubería («*pipe*») por la que circulen los datos. Estas conexiones se realizan mediante operadores (<< y >>) sobrecargados y funciones de E/S.

Un **flujo** es una abstracción desarrollada por Bjarne Stroustrup, el diseñador de C++, sobre la idea original de C y UNIX. Stroustrup se imaginó un flujo (corriente) de caracteres fluyendo desde el teclado a un programa, al igual que un flujo de agua fluye de un sitio a otro. Schwarz utilizó esta idea para crear la clase `istream`, una clase que representa un flujo de caracteres desde un dispositivo de entrada arbitrario a un programa en ejecución.

En esencia, un **flujo** es una abstracción que se refiere a una interfaz común a diferentes dispositivos de entrada y salida de una computadora. Existen dos formas de flujo: texto y binario. ***Flujos de texto*** se utilizan con caracteres ASCII, mientras que los ***flujos binarios*** se pueden utilizar con cualquier tipo de dato. Los sinónimos *extraer* u *obtener* se utilizan generalmente para referirse a la entrada de datos de un dispositivo e *inserción* o *colocación* (poner) cuando se refieren a la salida de datos a un dispositivo.

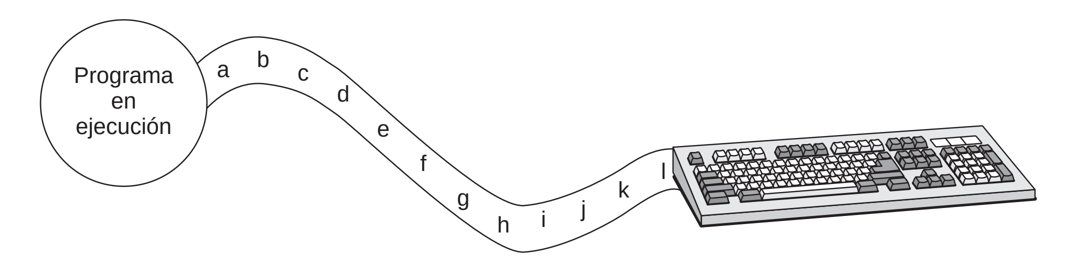


Figura 16.1. Simulación de un flujo.

16.1.1. Flujos de texto

Un *flujo de texto* es una secuencia de caracteres. En un flujo de texto, pueden ocurrir ciertas conversiones de caracteres si son requeridas por el entorno del sistema. Por ejemplo, un carácter de nueva línea puede convertirse en un par de caracteres *«retorno de carro/salto de línea*». Por esta razón, puede suceder que no se establezca una relación de uno a uno entre los caracteres que se escriben o se leen y los que aparecen en el dispositivo externo. De igual forma, como consecuencia de las posibles conversiones puede suceder que el número de caracteres escritos o leídos no coincida con el del dispositivo externo.

16.1.2. Flujos binarios

Un *flujo binario* es una secuencia de bytes que tiene una correspondencia uno a uno con los del dispositivo externo. Es decir, no se producen conversiones de caracteres. En este caso, la cantidad de bytes escritos o leídos coincide con la del dispositivo externo. Sin embargo, se puede añadir un número de bytes nulos definidos en la implementación, a un flujo binario. Estos bytes nulos, se suelen utilizar, por ejemplo, para ajustar la información de tal forma que se complete un sector de un disco.

> Un **flujo** (*stream*) es una secuencia de caracteres.

Hasta este momento del libro, la entrada y la salida se ha implementado utilizando el flujo de entrada estándar (denominado `cin`) y el flujo de salida estándar (denominado `cout`). Todo programa C++ tiene estos flujos disponibles automáticamente siempre que se incluya el archivo de cabecera *iostream.h*; `cin` y `cout` son objetos de tipos `istream` y `ostream` respectivamente. Los tipos definidos por el usuario `istream` y `ostream` están definidos en las clases de la biblioteca `iostream` y `cin/cout` se declaran como objetos de esos tipos. Normalmente, `cin` se conecta al teclado. La lectura de caracteres desde el objeto `cin` del flujo de entrada estándar es equivalente a la lectura del teclado; la escritura de caracteres al `cout` del flujo de salida estándar es equivalente a visualizar estos caracteres en su pantalla.

Todas las características de flujos entrada/salida se relacionan con conversiones de representaciones internas de datos (variables u objetos) a un flujo de caracteres (para salida) y conversión de flujos de caracteres a un formato interno correcto (para entrada). La Figura 16.2 muestra esta conversión para los flujos `cin` y `cout`. Mediante el operador de salida, <<, se convierten los datos que aparecen en el operando a un flujo de caracteres y «*se insertan*» en el flujo de salida (`cout`). Al contrario, el operador de entrada, >>, especifica «*la extracción del*» flujo de entrada de un flujo de caracteres. Estos caracteres se convierten entonces al formato interno apropiado y se almacenan en las posiciones de almacenamiento especificados.

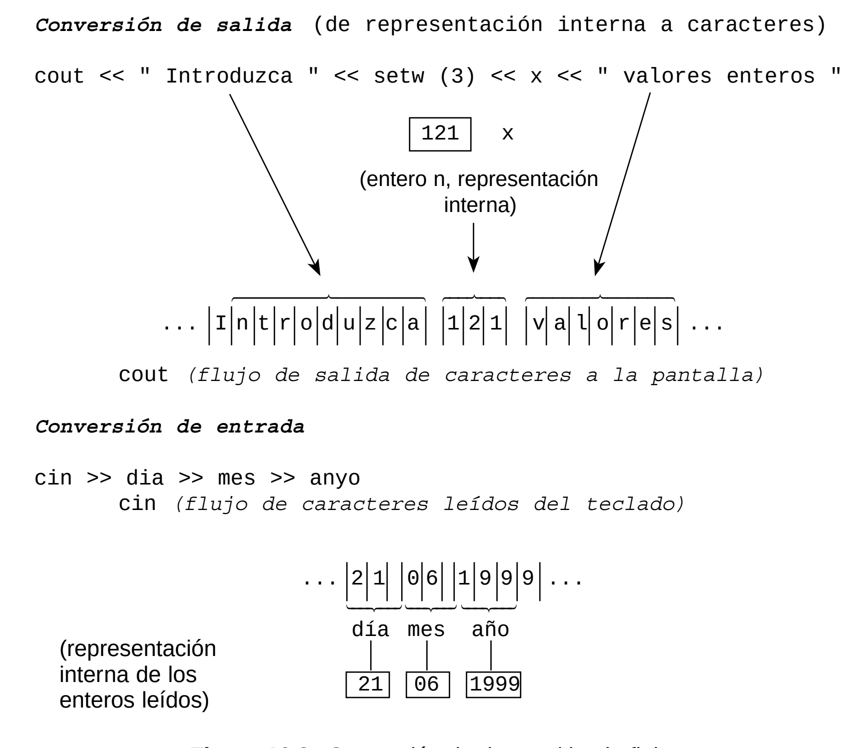


Figura 16.2. Conversión de datos a/desde flujos.

16.1.3. Las clases de flujo de E/S

El archivo de cabecera `<iostream>` declara tres clases para los flujos de entradas y salida estándar. La clase `istream` es para entrada de datos desde un flujo de entrada, la clase `ostream` es para salida de datos a un flujo de salida, y la clase `iostream` es para entrada de datos dentro de un flujo. Por otra parte estas clases declaran también los cuatro objetos ya conocidos (Tabla 16.1).

Tabla 16.1. Objetos de <iostream.h>.

Objeto de flujo	Función
`cin`	Un objeto de la clase `istream` conectado a la entrada estándar.
`cout`	Un objeto de la clase `ostream` conectado a la salida estándar.
`cerr`	Un objeto de la clase `ostream` conectado al error estándar, para salida sin búfer.
`clog`	Un objeto de la clase `ostream` conectado al error estándar, con salida a través de búfer.

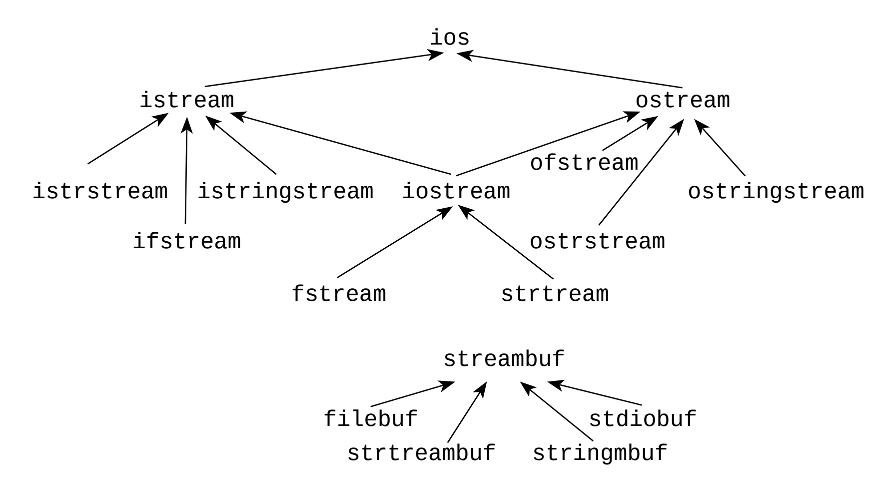


Figura 16.3. Biblioteca de clases de E/S.

Las clases que se derivan de la clase base `ios` se utilizan para procesamiento de flujos de alto nivel, mientras que las clases que se derivan de la clase base `streambuf` se utilizan para procesamiento de bajo nivel.

La clase `iostream` es la que se utiliza normalmente en operaciones ordinarias de E/S. Esta clase es una subclase de las clases `istream` y `ostream`, que a su vez son clases derivadas (subclases) de la clase base `ios`. Las tres clases que incluyen la palabra «`fstream`» en su nombre se utilizan para tratamiento de archivos y las cuatro clases con la palabra «`strstr`» en su nombre se utilizan para proceso de flujos de cadena en memoria. Por último, la clase `stdiobuf` se utiliza para combinar E/S de flujos C++ con las funciones antiguas de E/S estilo C.

16.1.4. Archivos de cabecera

Existen tres archivos de cabecera importantes para clases de flujos de E/S. El archivo de cabecera `<iostream>` declara las clases `istream`, `ostream` e `iostream` para las operaciones de E/S de flujos de entrada y salida estándar. Declara también los objetos `cout, cin, cerr` y `clog` que se utilizan en la mayoría de los programas C++.

El archivo de cabecera `<fstream>` declara las clases `ifstream, ofstream` y `fstream` para operaciones de E/S a archivos de disco. Por último, el archivo de cabecera `<strstream>` declara las clases `istrstream, ostrstream` y `strstream` para formateado de datos con búferes de caracteres.

Las clases de flujo C++ forman una jerarquía de clases (Fig. 16.3). En esta jerarquía se aprecian dos áreas principales: clases derivadas de `streambuf` y clases derivadas de `ios`.

El lenguaje de programación C++ no incluye facilidades de entrada/salida (E/S). Por esta razón se ha de utilizar la directiva

```
#include <iostream>
```

en cada programa que utilice E/S. El archivo de cabecera `iostream` incluye las definiciones de la biblioteca E/S.

16.2. LA BIBLIOTECA DE CLASES *IOSTREAM*

La biblioteca **`iostream`** se basa en el concepto de *flujos*; incorpora la ventaja de las potentes características orientadas a objetos de C++.

La biblioteca de E/S de flujos se construye a base de una jerarquía de clases que se declaran en diversos archivos de cabecera. La biblioteca de clases tiene dos familias paralelas de clases: las derivadas de `streambuf` y las derivadas de `ios`.

16.2.1. La clase `streambuf`

La clase `streambuf` proporciona un interfaz a dispositivos físicos; proporciona métodos fundamentales para realizar operaciones con *buffer* y manejo de flujos cuando las condiciones de *formateado* no son muy exigentes.

16.2.2. Jerarquía de clases `ios`

La jerarquía de clases `ios` gestiona todas las operaciones de E/S y proporciona el interfaz de bajo nivel al programador. La clase `ios` contiene un puntero a `streambuf`.

Para acceder a la biblioteca `iostream` se deben incluir archivos de cabecera específicos; uno de ellos ya ha sido utilizado por el lector, *`iostream`*, pero existen otros archivos de cabecera, como se verá en esta misma sección.

La clase `istream` (*`input stream`*) proporciona las operaciones de lectura de datos, mientras que la clase `ostream` (*`output stream`*) implementa las operaciones de escritura de datos. La clase `iostream` (*`input-output stream`*) se deriva simultáneamente de `istream` y `ostream`, y proporciona operaciones bidireccionales de entrada/salida (es un ejemplo de *herencia múltiple*).

Las clases `istrstream` y `ostrstream` se utilizan cuando se desea manejar arrays de caracteres y flujos, mientras que el otro conjunto de clases `istringstream` y `ostringstream`, se utilizan cuando se conectan flujos con objetos de la clase estándar `string`. La Figura 16.4 muestra el modo en el que se relacionan unas clases con otras.

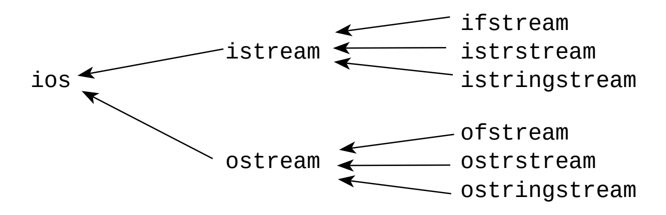


Figura 16.4. Clases derivadas de `ios`.

Las clases `ios`, `istream`, `ostream` y los objetos de flujos predeclarados (`cin`, `cout`, `cerr` y `clog`) se definen en el archivo `iostream`, el cual debe, por consiguiente, ser incluido en el programa. Las clases `ifstream` y `ofstream` se definen en el archivo de cabecera `fstream`. El archivo de cabecera `strstream` contiene las definiciones de las clases `istrstream` y `ostrstream` y por último las clases `istringstream` y `ostringstream` están declaradas en el archivo de cabecera `sstream`.

16.2.3. Flujos estándar

La biblioteca `iostream` define cuatro flujos estándar (objetos de flujo predefinidos): `cin`, `cout`, `cerr` y `clog` (Tabla 16.1) Estos tipos se declaran siempre automáticamente, de modo que no precisan declaración previa.

- El flujo `cin`, definido por la clase `istream`, está conectado al periférico de entrada estándar (el *teclado*, representado por el archivo `stdin`), aunque en algunos sistemas operativos podría ser redirigido (MS-DOS, Windows, Linux y UNIX).
- El flujo `cout`, definido por la clase `ostream`, está conectado al periférico de salida estándar (la *pantalla*, representado por el archivo `stdout`).
- El flujo `cerr`, definido por la clase `ostream`, está conectado al periférico de error estándar (la pantalla, representado por el archivo `stdout`). Este flujo no es a través de *buffer*.
- El flujo `clog`, definido por la clase `ostream`, está conectado igualmente al periférico de error estándar (la pantalla, representado por el archivo `stdout`). Al contrario que `cerr`, el flujo `clog` se realiza a través de *buffer*.

La ventaja de `cerr` sobre `clog` es que los *buffers* de salida se limpian (*vacían*) cada vez que `cerr` se utiliza, de modo que la salida está disponible más rápidamente en el dispositivo externo (que por defecto es la pantalla de vídeo). Sin embargo, en grandes cantidades de mensajes, la versión `clog` a través de *buffer* es más eficiente.

> Cualquier objeto creado de la clase `ios` o cualquiera de sus clases derivadas se conoce generalmente como un *objeto flujo*.

16.2.4. Entradas/salidas en archivos

Las tres clases siguientes permiten efectuar entradas/salidas en archivos:

- `ifstream`, clase derivada de `istream`; se utiliza para gestionar la lectura de un archivo. Cuando se crea un objeto `ifstream` y se especifican parámetros, se abre un archivo.
- `ofstream`, clase derivada de `ostream`; gestiona la escritura en un archivo. Los objetos `ofstream` se utilizan para hacer operaciones de salida de archivos. Se declara un objeto `ofstream` si piensa escribir un archivo de disco. Si se proporciona un nombre de archivo cuando se declara un objeto `ofstream`, se abre el archivo. Se puede especificar que el archivo se cree en modo binario o en modo texto. Si un objeto de `ofstream` está ya declarado, se puede utilizar la función miembro `open()` para abrir el archivo. Por otra parte, se dispone de la función miembro `close()`, que sirve para cerrar el archivo.
- `fstream`, clase derivada de `iostream`; permite leer y escribir en un archivo. Los objetos `fstream` se utilizan cuando se desea simultanear operaciones de lectura y escritura en el mismo archivo.

Las definiciones de estas clases se encuentran en el archivo de cabecera `fstream`.

16.2.5. Entradas/salidas en un *buffer* de memoria

Existen dos clases específicas destinadas a las entradas/salidas en un buffer en memoria:

- `istrstream`, clase derivada de `istream`, permite leer caracteres a partir de una zona de memoria, que sirve de flujo de entrada.

- `ostrstream`, clase derivada de `ostream`, permite escribir caracteres en una zona de memoria, que sirve de flujo de salida.

El archivo de cabecera `strstream` contiene las definiciones de las `clases istrstream` y `ostrstream`.

16.2.6. Archivos de cabecera

Cualquier programa que utilice la biblioteca iostream debe incluir el archivo de cabecera `<iostrem.h>`, y, eventualmente, otros archivos de cabecera suplementarios: `<ios>, <istream>, <ostream>, <ifstream>, <ofstream>, <fstream>, <strstream>, <iomanip>`.

El archivo de cabecera `io` declara clases de bajo nivel e identificadores. Los archivos `istream.h` y `ostream.h` soportan las entradas y salidas básicas de los flujos. El archivo `iostream.h` combina las operaciones de las clases en los dos archivos de cabecera anteriores. Para realizar entradas/salidas en archivos, se deben incluir los archivos de cabecera `fstream` e `iostream`. El archivo `iomanip` permitirá *formatear* y organizar la salida de datos. La inclusión del archivo de cabecera `strstream` permite el acceso a las funciones de la biblioteca `iostream`, que permiten efectuar las entradas/salidas en memoria.

16.2.7. Entrada/salida de caracteres y flujos

C++ visualiza todas las entradas y salidas como flujos de caracteres. Si su programa obtiene la entrada del teclado, un archivo de disco, un módem o un ratón, C++ ve sólo un flujo de caracteres. C++ no conoce cuál es el tipo de dispositivo que le proporciona la entrada.

Estas operaciones de E/S de flujos significan que se utilizan las mismas funciones para obtener la entrada del teclado como del módem. Se pueden utilizar las mismas funciones para escribir en un archivo de disco, una impresora o una pantalla. Naturalmente, se necesita algún medio para encaminar el flujo de entrada o salida al dispositivo adecuado.

El flujo de datos irá de un dispositivo de entrada (teclado) al programa C++ y de un programa C++ al dispositivo de salida (la pantalla o la impresora).

16.3. CLASES `ISTREAM` Y `OSTREAM`

Las clases `istream` y `ostream` se derivan de la clase base `ios`.

```
class istream: virtual public ios { // ... };
class ostream: virtual public ios { // ... };
```

16.3.1. Clase `istream`

La clase `istream` permite definir un flujo de entrada y soporta métodos para entrada *formateada* y *no formateada*. El operador de extracción >> está sobrecargado para todos los tipos de datos integrales de C++, haciendo posible operaciones de entrada de alto nivel.

Declaración

```
class istream: virtual public ios      // iostream
{
   // ...
};
```

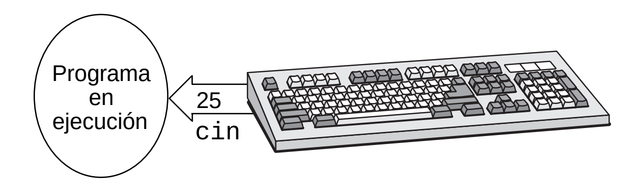
Un *objeto de flujo* es una instancia de una subclase de la clase `ios`. El operador de entrada >> se aplica a un objeto `istream` y a un objeto variable.

```
objeto_istream >> objeto_variable
```

y trata de extraer una secuencia de caracteres correspondientes a un valor del tipo de `objeto_variable` de `objeto_istream`. Si no hay caracteres se bloquea la ejecución precedente hasta que se introducen caracteres. Como ejemplo, supongamos que `cin` está inicialmente vacío y se ejecutan las sentencias siguientes:

```
int edad;
cin >> edad;
```

Si el usuario introduce `25`, los caracteres `2` y `5` se introducen en el objeto `cin` de `istream`. Se leen los caracteres `2` y `5` de `iostream`, los convierte al valor entero `25` y almacena este valor en su operando derecho `edad`.



> El operador >> se suele denominar *operador de extracción* debido a que su comportamiento es extraer valores de una clase `istream`.

Indicadores de estado

¿Qué sucederá si el usuario introduce valores no apropiados? Por ejemplo, en el caso anterior, en lugar de escribir `25` se comete un error y se teclea `t5`. Entonces el flujo `cin` contendrá los caracteres `t` y `5`. Es decir, el operador >> trata de extraer un entero y encuentra el carácter `t`.

La clase `istream` tiene un atributo que es su **estado** o **condición**. El estado se representa mediante *indicadores* o *banderas* (*«flags»*) que representan la condición o estado de un flujo (Tabla 16.2). Las tres condiciones básicas son: `good` (el estado es bueno); `bad` (hay algo incorrecto en el flujo); `fail` (la última operación del flujo no tiene éxito). La clase `istream` contiene una variable booleana llamada **bandera** para cada uno de estos estados, con el indicador `good` inicializado a `true` (verdadero) y los indicadores `bad` y `fail` inicializados a falso.

En el ejemplo anterior, al encontrar la letra `t` cuando se intentaba leer un entero, se pone el indicador `good` del flujo a falso, el indicador `bad` a `true` (verdadero) y `fail` también verdadera, y viceversa. Para cada uno de estos indicadores, la clase `istream` proporciona una función miembro `boolean` que tiene el mismo nombre que su indicador, que informa del valor de ese indicador.

Tabla 16.2. Indicadores de estado.

Llamada a función	**Devuelve *true* si y sólo si**
`cin.good ()`	Todo está correcto en `istream`
`cin.bad ()`	Algo está mal en el `istream`
`cin.fail ()`	No se puede completar la última operación

Variables de estado de `ios`

Cada clase `ios` tiene unas variables de estado que se especifican en la definición `enum`.

```
class ios {
public:
   enum {           // valor del indicador de estado de error
      goodbit = 0,  // todos correctos
      eofbit  = 01, // fin de archivo
      failbit = 02, // última operación fallada
      badbit  = 04  // operación no válida
   };
   // otros miembros incluidos aquí
};
```

Los indicadores de formato de flujo sólo se pueden cambiar explícitamente y sólo mediante las funciones de acceso de `ios`. En contraste, variables de estado de flujo se cambian implícitamente, como resultado de operaciones de E/S. Por ejemplo, cuando un usuario introduce `Control-D` (o `Control-Z` en computadoras DOS y VAX) para indicar el final del archivo; el *indicador de estado eof* de `cin` se pone a 1, y se dice que el flujo está en un estado *eof*.

Las cuatro *variables de estado* (`goodbit, failbit, eofbit` y `badbit`) pueden ser accedidas individualmente por sus funciones de acceso (`good(), fail(),` y `bad()`). Puede accederse también a todas ellas colectivamente mediante la función `rdstate()`.

Ejemplo 16.1

```
main()
{
   cout << "cin.rdstate() = " << cin.rdstate << endl;
   int n;
   cin >> n;
   cout << "cin.rdstate() = " << cin.rdstate() << endl;
}
```

Ejecución

```
cin.rdstate() = 0
22
cin.rdstate() = 0
```

Segunda ejecucción

```
cin.rdstate() = 0
^D control-D
cin.rdstate() = 3
```

En la segunda ejecución, el usuario pulsó ***Control-D*** para señalar el final del archivo. Esta acción fija a `eofbit` y `failbit` de `cin`, que tienen valores numéricos 1 y 2, toman el valor total 3 de la variable de estado `_state`.

***La función* `clear()`**

El uso de **Control-D** (o **Control-Z**) para terminar la entrada es sencillo y adecuado. Pulsando esta secuencia de teclas, se fija el valor de `eofbit` en el flujo de entrada. Pero a continuación, si se desea uti-

lizarlo de nuevo en el mismo programa, se ha de borrar (limpiar) primero. Esta acción se realiza con la función `clear()`.

Ejemplo 16.2

```
int main()
{
   int n, suma = 0;
   while (cin >> n)
      suma += n;
   cout << "La suma parcial es " << suma << endl;
   cin.clear();
   while (cin >> n)
      suma += n;
   cout << "La suma total es" << suma << endl;
   return 0;
}
```

Ejecución

```
30    20   50    ˆD
La suma parcial es 100
40    80
La suma total es 220
```

16.3.2. La clase `ostream`

La clase `ostream` permite a un usuario definir un flujo de salida y soporta métodos para salidas *formateadas* y *no formateadas*. El operador de inserción se sobrecarga para todos los tipos de datos integrales. Las clases `ofstream, ostrstream, constream, iostream` y `ostream_withassign` se derivan todas de `ostream`.

Al igual que `istream`, la clase `ostream` se deriva virtualmente de la clase `ios` para evitar declaraciones múltiples cuando se declara `iostream`.

```
class ostream: virtual public ios   // iostream
{
   // ...
};
```

La noción abstracta de un flujo se puede utilizar para ocultar estos detalles de bajo nivel del programador, y de este modo la biblioteca `iostream` proporciona la clase `ostream` para representar el «flujo» de caracteres de un programa en ejecución a un dispositivo de salida arbitrario (véase la Figura 16.5).

Después de crear una clase `ostream`, se definen dos objetos `ostream` de la biblioteca `iostream` de modo que cualquier programa que incluye el archivo de cabecera de la biblioteca tiene dos flujos de salida del programa a cualquier dispositivo que el usuario esté utilizando para salida: una ventana, un terminal, etc.

1. `cout`, un `ostream` para visualizar salida normal.
2. `cerr`, un `ostream` para visualizar mensajes de error o de diagnóstico.

El mecanismo `assert()` normalmente escribe sus mensajes de diagnóstico a `cerr`, no a `cout`.

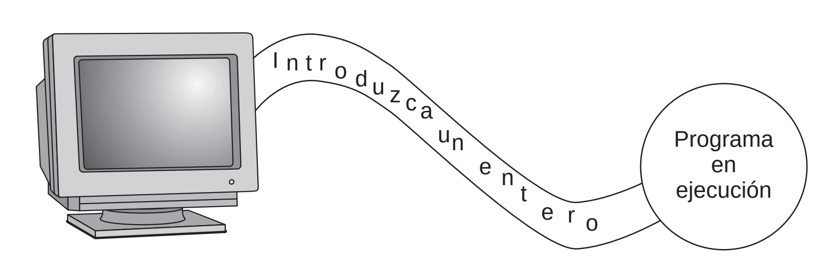


Figura 16.5. Simulación de flujo `ostream`.

El operador <<

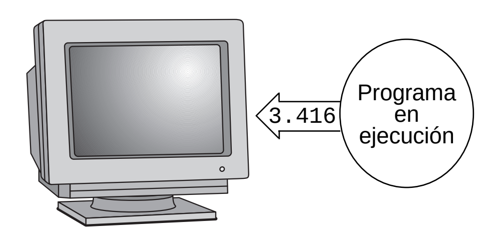
Cuando el operador << se aplica a un objeto `ostream` y una expresión

`objeto_ostream << expresión`

se evaluará la expresión y se inserta la secuencia de caracteres correspondientes a ese valor en el objeto `ostream`. Por consiguiente, en el caso de:

```
const double PI = 3.1416;
cout << PI;
```

la función << convierte el valor `double, 3.1416,` en los caracteres correspondientes `3, ., 1, 4, 1` y `6` y los inserta uno a uno en `cout`:



Los caracteres permanecen realmente en el `ostream`, sin aparecer en la pantalla, hasta que se *limpia* (*fluye*) el `ostream`, que, como su nombre sugiere, vacía el flujo en la pantalla.

Ejemplo 16.3

```
double f(double x, double y)
{
   cout << "Introducción de f";
      .
      .
      .
   cout << "salida de f";
   return z;
}
```

La salida `"Introducción de f"` puede no aparecer en nuestra pantalla cuando se espera ya que el `ostream` no se ha limpiado.

Manipulador de salida

Un medio común para limpiar un `ostream` es utilizar un **manipulador de salida** —un identificador que afecta al propio `ostream` cuando se utiliza en una sentencia de salida, en lugar de generar simplemente un valor que aparece en la pantalla. El manipulador más utilizado para limpiar un flujo de salida es el **manipulador** `endl`:

```
double f(double x, double y)
{
   cout << "Introducción de f" << endl;
      .
      .
      .
   cout << "salida de f" << endl;
   return z;
}
```

Este manipulador inserta un carácter de nueva línea (`'\n'`) en el `ostream` y lo limpia, terminando una línea de salida.

Otro manipulador utilizado es `flush`. Este manipulador simplemente limpia el `ostream` sin insertar nada:

```
double f(double x, double y)
{
   cout << "Introducción de f" << flush;
      .
      .
      .
   cout << "Salida de f" << flush;
   return z;
}
```

16.4. SALIDA A LA PANTALLA Y A LA IMPRESORA

Cuando se ejecutan los programas C++ normalmente se deseará generar información en uno de los dos dispositivos hardware típicos: un monitor o una impresora. De hecho, habrá ocasiones en que se necesitará generar información en ambos dispositivos durante la ejecución de un programa.

Se puede escribir información en el dispositivo de salida utilizando el objeto `cout`, que está definido en el archivo de cabecera `iostream`. Los objetos son el núcleo de la programación orientada a objetos y su uso correcto en C++ potenciará sus programas.

El método más común de dirigir la salida a la pantalla es utilizar el objeto `cout`. El flujo de salida que se pasa al objeto `cout` se dirige al flujo de salida estándar. Mediante el operador de inserción (`<<`) se ponen datos en el flujo de salida.

El operador `<<` dirige el contenido de la variable a su derecha al objeto (`cout`) de su izquierda. Así,

```
cout << "Hola mundo, C++";
```

El operador de inserción se define para todos los tipos de datos básicos: `char, unsigned char, signed char, short, unsigned short, int, unsigned int, long, unsigned long, float, double, long double, void*` y `char*` (un puntero a una cadena). El operador de in-

serción convierte los datos a la derecha de `<<` a una cadena de caracteres (`char`), tipo esperado por el objeto `cout`. Por ejemplo, el siguiente valor entero, `val_ent`, se convierte a la cadena `47` y se pasa al objeto `cout`:

```
int val_ent = 47;
cout << val_ent;
```

En el caso anterior se visualiza `47`. La variable `val_ent` es una expresión, por lo que podría también ser válida la sentencia:

```
int i, j;
cout << i+j;
```

16.4.1. Operadores de inserción en cascada

El operador de inserción se puede poner en cascada, de modo que pueden aparecer varios elementos en una sola sentencia C++. Así, la sentencia

```
cout << 1 << 2 << 3 << 4;
```

generará una salida tal como

```
1 2 3 4
```

Si se desea escribir información de caracteres, se debe encerrar la información de salida entre comillas. La sentencia

```
cout << "Hola, programador de C++";
```

genera la salida

```
Hola, programador de C++
```

Los operadores en cascada pueden mezclar valores de caracteres y numéricos. Así, por ejemplo,

```
cout << "Total =" << Suma << endl;
```

visualizará

```
Total = 450
```

Suponiendo que el valor de la variable `Total` es `450`. El símbolo `endl`, como ya conoce el lector, hace que el flujo de salida avance a la siguiente línea. Se puede situar `endl` en cualquier parte del flujo, aunque usualmente se sitúa al final de la línea. La sentencia siguiente:

```
cout   << "Los resultados son los siguientes:" << endl;
cout   << "Enero   " << Total_Ene << endl;
cout   << "Febrero " << Total_Feb << endl;
cout   << "Marzo   " << Total_Mar << endl;
```

produce la salida

```
Los resultados son los siguientes:
Enero    300
Febrero  425
Marzo    106
```

Desde el punto de vista práctico, cada operador de inserción envía un dato (una constante, una expresión o una variable) al flujo de salida, y se pueden concatenar datos de tipos diferentes en una única expresión `cout`.

```
cout << Voltaje << Corriente << Resistencia;
```

La sentencia precedente escribirá los valores almacenados en memoria en el mismo orden en que están escritos. El orden de salida será el mismo que el orden listado en la sentencia `cout`, pero sin ningún espacio entre los valores. Si desea espaciado entre los valores deberá insertar caracteres en blanco dentro de la sentencia `cout`.

16.4.2. Las funciones miembro `put()` y `write()`

Las clases definen datos y funciones miembro. Una función de una clase se llama un *método* o *función miembro*. La clase `ostream` proporciona la función **`put()`** para insertar un único carácter en el flujo de salida y la función **`write()`**, para insertar una cadena en el flujo de salida. Ambas funciones devuelven un objeto `cout`.

Se puede escribir en un flujo de salida llamando a las funciones miembro **`put()`** o **`write()`**. Los formatos de estas funciones son:

```
dispositivo.put(valor_caracter);
dispositivo.write(valor_cadena, num);
```

El punto que separa *dispositivo* de la función **`put()`** es el operador de miembro (**.**). *`valor_caracter`* puede ser una constante, expresión o variable carácter (`char`) y *`valor_cadena`* una cadena; *num* es un valor `int` utilizado para especificar el número de caracteres de la cadena a visualizar. El *`dispositivo`* puede ser cualquier dispositivo de salida estándar. Para escribir, por ejemplo, un carácter en su impresora, se abrirá `PRN` con `ofstream`.

Así, las sentencias siguientes visualizan dos caracteres (`'Z'` y `'l'`) en el flujo de salida:

```
cout.put('Z');
char letra = 'l';
cout.put(letra);
```

Si desea escribir un bloque de caracteres, utilice la función miembro `write`, como en estos ejemplos:

```
cout.write("Biblioteca", 3);      // Se visualiza Bib

cout.write("Día Nacional", 12) << "\n";
cout.put(65) << "Antonio Molina \n";
cout.put('H').write("ola", 4) << " mundo C++ \n";
```

Al ejecutarse las sentencias anteriores se visualiza

```
Día Nacional
Antonio Molina
Hola mundo C++
```

Obsérvese que la primera función **`put()`** contiene una constante de valor entero; el entero se interpreta como un código ASCII de la letra `A` que se visualiza.

La función **`write()`** visualiza tantos caracteres como se especifique en el segundo argumento. Si se especifica un número mayor que el número de caracteres de la cadena, la función visualiza cualquier cosa que resida en memoria a continuación de la cadena.

16.4.3. Impresión de la salida en una impresora

El envío de la salida de un programa a la impresora es fácil con la función `ofstream`. El formato de `ofstream` es

`ofstream` *`dispositivo (nombre_dispositivo)`*

y su uso requiere el archivo de cabecera `fstream.`

Ejemplo 16.4

El siguiente programa solicita al usuario el nombre y apellidos. A continuación, imprime el nombre completo y el apellido en la impresora.

```
// SALIMPRE.CPP
// Imprime un nombre en la impresora

#include <fstream.h>
using namespace std;

int main()
{
   char nombre[20];
   char apellidos[30];

   cout << "¿Cuál es su nombre?";
   cin >> Nombre;
   cout << "¿Cuáles son sus apellidos?";
   cin >> Apellidos;

   // Enviar nombre y apellidos a la impresora

   ofstream impresora ("PRN");
   impresora << "Su nombre completo es: \n";
   impresora << apellidos << ", " << nombre << endl;

   return 0;
}
```

16.5. LECTURA DEL TECLADO

Obtener información en un programa para su procesamiento se denomina *lectura*. En la mayoría de los sistemas actuales, la información se lee de una de las dos fuentes: de un teclado o de un archivo de disco. En esta sección, aprenderá cómo se lee la información procedente del teclado.

La sentencia C++ que se utiliza para la lectura de datos del teclado es `cin`. Al igual que `cout`, `cin` es un objeto predefinido en C++ y es parte del archivo de cabecera `iostream`.

Cuando se teclea, se genera un flujo de entrada. Al igual que con la salida, C++ utiliza un enfoque orientado a objetos para la entrada. El objeto `cin` extrae caracteres del flujo de entrada, lo convierte a cualquier tipo de dato diseñado en la sentencia de entrada y lo almacena en la posición de memoria deseada.

Se utiliza el operador de extracción >> para manejar la entrada de un flujo. El operador obtiene datos del flujo y lo sitúa en una variable.

Un ejemplo de una sentencia de entrada en C++ es:

```
int valor;
cin >> valor;
```

El operador de extracción se utiliza con el objeto `cin` para introducir datos desde el teclado. El operador >> es fácil de recordar, ya que sugiere un flujo de datos desde la izquierda a la derecha. El operador se denomina operador de extracción, ya que *extrae* datos desde el flujo de entrada.

Al igual que el operador de inserción, el operador de extracción se define para todos los tipos de datos básicos: `char, unsigned char, signed char, short, unsigned short, int, unsigned int, long, unsigned long, float, double, long double`, cadenas y punteros. El operador de extracción convierte los datos desde el flujo de entrada al tipo de datos esperado por la variable que recibe el dato.

El operador de extracción se suele utilizar en unión con un operador de inserción y un mensaje de petición de datos o salutación:

```
int Edad;
cout << "Introduzca edad del alumno:";
cin >> Edad;
```

Al igual que el operador de inserción, el operador de extracción se puede poner en cascada, con un formato similar a:

```
cin >> variable1 >> variable2 >> ... >> variablen
```

Ejemplo 16.5

```
int Edad;
float Altura;
cout << "Introduzca Edad y Altura:";
cin >> Edad >> Altura;
```

Cuando se procesan elementos múltiples en la entrada, se debe teclear al menos un espacio en blanco entre los elementos de entrada. La entrada a las sentencias anteriores podría ser:

```
Introduzca Edad y Altura: 47 75
```

Ejemplo 16.6

Introduzca un valor entero desde el teclado y, a continuación, visualícelo.

```
// ENDATOS1.CPP - Introducir un número utilizando el objeto
// cin
```

```
#include <iostream>
using namespace std;

void main()
{
   int val_e;
   cout << "Introduzca un número:";
   cin >> val_e;
   cout << "\n Ha introducido un número" << val_e << endl;
}
```

Al ejecutarse el programa anterior, se podrían seguir estas acciones:

```
Introduzca un número: 4321<Intro>
Ha introducido el número 4321
```

> La notación <*Intro*> se utiliza para representar la pulsación de la tecla `INTRO (ENTER O RETURN).`

16.5.1. Lectura de datos carácter

Los caracteres se leen uno a uno, de acuerdo con las reglas siguientes:

1. Los *blancos* (espacios en blanco, tabulaciones, nuevas líneas y avances de página) son ignorados por `cin` cuando se utiliza el operador >>. Sin embargo, los espacios en blanco se pueden leer utilizando un operador `cin` diferente.
2. Los valores numéricos se pueden leer como caracteres, pero cada dígito es un carácter independiente.

```
// Este programa lee datos carácter

#include <iostream>
using namespace std;

void main()
{
   char Letra1;
   char Letra2;
   char Letra3;
   cin >> Letra1 >> Letra2 >> Letra3;
   cout << Letra1 << Letra2 << Letra3;
}
```

Algunas ejecuciones del programa anterior son:

```
Letras            ABC         75 47 5        47.543
Let               ABC         754            47.
```

Obsérvese que la lectura de un carácter cada vez impone una restricción en la introducción de datos carácter. Se requiere una variable independiente para cada carácter individual introducido. Por esta causa, se necesita disponer de un medio para leer datos de cadena.

16.5.2. Lectura de datos cadena

Cuando se utiliza el operador de extracción para leer datos tipo cadena, se producirán anomalías cuando las cadenas consten de más de una palabra separada por blancos.

Ejemplo 16.7

```
// Listado LEERCAD1.CPP

#include <iostream>
using namespace std;

int main()
{
   char nombre[30];
   int edad;
   float salario;

   cout << "\n Introduzca nombre, edad y salario: \n";

   cin >> nombre >> edad >> salario;
   cout  << "\n Nombre:   "
         << nombre
         << "\n Edad:      "
         << edad
         << "\n Salario:   "
         << salario
         << endl;
   return 0;
}
```

Al ejecutarse el programa se puede producir una salida tal como

```
Mortimer 47 350500<Intro>
Nombre:      Mortimer
Edad: 47
Salario:     350500
```

Si la cadena de entrada contiene más de una palabra, el objeto `cout` no leerá más que la primera palabra, truncando el resto de la cadena.

```
// Este programa muestra cómo cin lee datos cadena
// mediante el operador >>

#include <iostream>
using namespace std;

void main()
{
   char Nombre[30];

   cin >> Nombre;
   cout << '\n' << Nombre;
}
```

Cuando el usuario teclea

```
Pepe Mackoy
```

el sistema visualiza

```
Pepe
```

La razón de los caracteres truncados (`Mackoy`) es que cuando se leen datos cadena, el operador `>>` hace que el objeto `cin` termine la operación de lectura, siempre que se encuentre cualquier espacio en blanco, de modo que la variable `Nombre` contiene sólo `Pepe`.

El sistema para resolver esta anomalía puede ser definir una cadena para cada palabra completa a introducir. Sin embargo, el método más eficiente consistirá en utilizar funciones miembro **`get()`** y **`getline()`**.

16.5.3. Funciones miembro `get()` y `getline()`

El objeto flujo `cin` contiene varias funciones miembro que permiten procesar entrada de cadenas y caracteres sin utilizar el operador de extracción. La función miembro **`get()`** lee un único carácter o una línea de datos del teclado (cadenas).

La función miembro `get()` tiene varios formatos y está definida en la clase `istream`. Desde un punto de vista práctico, `get()`se puede utilizar de dos modos: `get(),` se utiliza sin parámetros en cuyo caso el valor devuelto se puede utilizar con una referencia a un carácter; otro formato es `get` con parámetros (una referencia a un carácter).

El primer formato de `get()` es sin parámetros. Este formato devuelve el valor del carácter encontrado y devolverá `EOF` (fin de archivo, *«end of file»*) si se alcanza el final del archivo. Su prototipo de función es:

```
int get()
```

Esta versión se suele utilizar normalmente en bucles de entrada.

Ejemplo 16.8

Lectura de caracteres con la función `cin.get()`.

```
#include <iostream>
using namespace std;

int main()
{
   char c;
   while ((c = cin.get())!= EOF)
   {
      cout << "c:" << c << endl;
   }
   cout << "\n Terminado!\n";
   return 0;
}
```

> ### *Nota*
>
> Para salir de este programa, debe enviar un final de archivo desde el teclado. En computadoras DOS/Windows utilice *`Ctrl+z`*; en sistemas UNIX utilice *`Ctrl+D`*.

Al ejecutar el programa se produce esta salida.

```
Hola
c : H
c : o
c : l
c : a

Mundo
c : M
c : u
c : n
c : d
c : o

(Ctrl-z)          // o bien ^D, Ctrl+D
Terminado!
```

Explicación

Cada llamada de la función `cin.get()`lee un carácter más de `cin` y lo devuelve a la variable `c`. A continuación, la sentencia del interior del bucle inserta `c` en el flujo de salida. Estos caracteres se acumulan en un búfer hasta que se inserta el carácter *fin de línea*. Entonces se limpia el búfer. Cuando se encuentra `EOF` (*`Ctrl+z`* o *`Ctrl+D`*), se sale del bucle. En la mayoría de las computadoras `EOF` toma el valor –1.

Ejemplo 16.9

La constante entera `EOF`

```
void main()
{
   cout << "EOF =" << endl;
}
```

Al ejecutarse

```
EOF = -1
```

Otro formato de la función `get()` lee el siguiente carácter del flujo de entrada en su parámetro que se pasa por referencia

```
istream& get(char& c);
```

Esta versión devuelve *nul* cuando se detecta el final del archivo, de modo que se puede utilizar para controlar un bucle de entrada tal como éste:

```
while (cin.get(car))
```

Ejemplo 16.10

Lectura de caracteres con la función `cin.get()`.

```
main()
{
   char c;
   while (cin.get(c))
      cout << c;
   cout << endl;
}

En un lugar de Sierra Magina
En un lugar de Sierra Magina
Conocido por Carchelejo
Conocido por Carchelejo
^D
```

Ejemplo 16.11

Otra aplicación de `get()` con parámetro referencia.

```
1:      // Listado get()
2:      #include <iostream>
3:      using namespace std;
4:
5:      int main()
6:      {
7:         char a, b, c;
8:
9:          cout << "Introduzca tres letras:";
10:         cin.get(a).get(b).get(c);
11:
12:         cout << "a:" << a << "\nb:" << b << "\nc:" << c << endl;
13:         return 0;
14:   }
```

Ejecución

```
Introduzca tres letras: ono
a: o
b: n
c: o
```

Explicación

En la línea 6 se crean tres variables de carácter. En la línea 9, se llama tres veces a `cin.get()`... y se pone la primera letra en `a` y vuelve a `cin`, de modo que se llama a `cin.get(b)` y se pone la siguiente letra en `b`. El resultado final es que se llama a `cin.get(c)` y se pone la tercera letra en `c`.

Existe un tercer formato de la función `get()` —similar a la función `getline()` que se verá posteriormente. Su prototipo es

```
istream& get (char* bufer, int n, char sep = '\n');
```

Este formato lee caracteres del *búfer* hasta que se lean, o bien *n-1* caracteres, o bien hasta que se encuentre el carácter separador *sep*.

> ***Regla del prototipo de `get()`***
>
> El primer parámetro es un puntero a un array de caracteres; el segundo parámetro es el número máximo de caracteres a leer más uno y el tercer parámetro es el carácter de terminación.

Ejemplos

```
char cadena[80];          // Array de entrada
cin.get(cadena, 80);      // Lectura de caracteres hasta que
                          // se encuentra un carácter nueva
                          // línea o se
                          // han leído 79 caracteres
```

Ejemplo 16.12

```
Lectura de caracteres con cin.get()
main()
{
   char bufer[80];
   cin.get(bufer, 8);      // lee 7 caracteres del bufer
   cout << "[" << búfer << "]\n";
   cin.get(bufer, sizeof(bufer));
   cout << "[" << búfer << "]\n";
}
```

Si el búfer almacena `ABCDE|FGHIJ|KLMNO|PQRST|UVWXY|Z`
la salida será:

```
[ABCDE|F]
[GHIJ|KLMNO|PQRST|UVWXY|Z]
```

16.5.4. La función `getline`

Otra función miembro que se puede utilizar para la lectura de datos es **`getline()`**. La función **`getline()`** permite a `cin` leer cadenas completas, incluyendo espacios en blanco; es muy similar a la función miembro `get()` de dos o tres argumentos, excepto que el carácter de terminación en **`getline()`** se extrae del flujo de entrada y se considera. El formato de `getline()` es

```
cin.getline(var_cad, long_max_cad+2, car_separador);
```

La función **`getline()`** utiliza tres argumentos. El primer argumento, *`var_cad`*, es el identificador de la variable cadena. El segundo argumento es la máxima longitud de la cadena (número máximo de caracteres que se leen); la longitud ha de ser mayor que la cadena real al menos en dos caracteres, para permitir los caracteres `'\n'` (*CRLF*) y `'\0'` (carácter nulo). El carácter de separación se lee y almacena como el siguiente al último carácter de la cadena. La función **`getline()`** inserta automáticamente el carácter de terminación *nulo* como el último carácter de la cadena. Si no se especifica ningún carácter de terminación, se toma por defecto el carácter `'\n'`.

Veamos un programa ejemplo:

```
// Uso de cin y getline para leer datos de cadena

#include <iostream>
using namespace std;

void main()
{
   char Nombre[40];
   cin.getline(Nombre, 40);
   cout << Nombre;
}
```

Al ejecutar el programa con la entrada `Sierra de Cazorla` la variable `Nombre` acepta toda la cadena, es decir, todos los caracteres incluyendo blanco.

Ejemplo 16.13

```
// Este programa lee y escribe
// Nombre, Dirección y Teléfono del usuario

#include <iostream>
using namespace std;

void main()
{
   //Definición de arrays de caracteres
   char Nombre[40];
   char Calle[30];
   char Ciudad[30];
   char Provincia[30];
   char CodigoPostal[5];
   char Telefono[10];

   //Lectura de datos
   cin.getline(Nombre, 40);
   cin.getline(Calle, 30);
   cin.getline(Ciudad, 30);
   cin.getline(Provincia, 30);
   cin.getline(CodigoPostal, 5);
   cin.getline(Telefono, 10);

   //Visualizar datos
   cout << Nombre;
   cout << Calle;
   cout << Ciudad;
   cout << Provincia;
   cout << CodigoPostal;
   cout << Telefono;
}
```

Una entrada de datos en una ejecución del programa:

```
Luis Enrique
Santiago Bernabéu 45
Madrid
Madrid
28230
91-7151515
```

producirá una salida tal como:

```
Luis Enrique
Santiago Bernabéu 45
Madrid
Madrid
28230
91-7151515
```

> La diferencia entre `get()` y `getline()` es que esta última almacena el carácter separador o delimitador en la cadena antes de añadir el carácter nulo.

16.5.5. Problemas en la utilización de `getline()`

Aunque `getline()` funciona cuando se leen datos tipo de cadena de modo consecutivo, se presentarán problemas cuando se intenta utilizar una variable de cadena, después de que se ha utilizado `cin` para leer una variable carácter o numérica. Por ejemplo, supongamos un programa como el siguiente:

```
// PERGETL.CPP
// Programa que muestra el problema de utilizar cin.getline()
// para leer una cadena después de haber leído una
// variable numérica o carácter

#include <iostream>
using namespace std;

void main()
{
   char Nombre[30];
   int Edad;

   cout << "Introduzca edad:";
   cin  << Edad;
   cout << Edad;

   cout << "Introduzca el nombre:";
   cin.getline(Nombre, 30);
   cout << Nombre;
}
```

Si se introducen en `Edad` y `Nombre` los valores `525` y `Mortimer`; es decir, suprimiendo la salida del programa siguiente:

```
Introduzca edad: 525
Introduzca el nombre: Mortimer
```

Los valores que toman las variables citadas son:

```
Edad 525
Nombre '\n''\0'
```

La razón de los valores anteriores se debe a que al introducir `525`, se debe pulsar la tecla `<INTRO>(ENTER)` a la terminación. Esta acción inserta un carácter `CRLF` (retorno de carro/avance de línea) y permanece en el *buffer* (memoria intermedia). Cuando la sentencia `cin.getline(Nombre, 30)` se ejecuta, se lee la memoria intermedia del teclado y se encuentra el carácter `CRLF`, dado que éste es, por defecto, el carácter de separación, se detiene la lectura y se inserta el carácter de terminación nulo en el *array*. Por consiguiente, no se puede introducir el nombre.

Existen tres métodos para resolver el problema:

1. Especificar un carácter de separación diferente en la función `getline()`. El usuario debe introducir este carácter y se almacenará como último carácter, antes del carácter nulo del array.
2. Limpiar la memoria intermedia (*buffer*) del teclado leyendo el carácter sobrante `CRLF` en una variable *basura*, después de leer cualquier dato numérico o carácter y antes de leer cualquier dato cadena. De este modo en la variable basura (auxiliar) se almacenarían los datos de la memoria intermedia y ya se podrían leer los caracteres útiles para la variable cadena.

```
// LIMPIARB.CPP
// Limpieza de la memoria intermedia con una
// variable auxiliar o basura

#include <iostream.h>
using namespace std;

void main()
{
   char Auxiliar[2];
   char Nombre[30];
   int Edad;

   cout << "Introduzca edad:";
   cin >> Edad;                    // Lectura de datos numéricos
   cout << Edad;
   cin.getline(Auxiliar, 2);       // Limpiar buffer de teclado
   cout << "Introduzca el nombre:";
   cin.getline(Nombre, 30);        // Leer datos de cadena
   cout << Nombre;
}
```

3. Utilizar una sentencia de lectura diferente; para ello se recurre a funciones de cadena de E/S definidas en la biblioteca estándar de C y C++: **`gets()`** y **`fgets()`**. Estas funciones se encuentran dentro del archivo de cabecera *stdio.h*.

16.6. FORMATEADO[1] DE SALIDA

Si no se instruye al operador de inserción para realizar operaciones de formateado específico, la salida se formatea cuando se convierte un dato a un flujo de caracteres para la salida. La Tabla 16.3 describe cómo formatea la salida el operador de inserción para diversos tipos de datos.

Ejemplos

```
cout << "CAZORLA#";
char letra = 'J';
cout << letra;

float f = 123.456789101, g = 1234567890.456;
cout << f << '#\n';
cout << g << '#\n';
int e = 123, en = -525;
cout << e << en << '\n';
```

Si se ejecutan las sentencias `cout` precedentes se produce una salida tal como ésta:

```
CAZORLA#J123.456789#
1.234567E+009#
123-525
```

Tabla 16.3. Conversión para salida de tipos de datos.

Tipo	**Tipo de conversión de salida**
`char`	Los caracteres imprimibles se visualizan con una anchura de una columna. Los caracteres de control, tales como nueva línea, tabulación, etc., pueden producir más caracteres de salida.
`int`	Cualquier tipo entero (`int, short o long`) se visualizan como números decimales con anchura suficiente para contener el número y un signo menos si el entero en negativo.
Cadena	La anchura en pantalla es igual a la longitud de la cadena.
`float`	Los números reales en coma flotante se visualizan con seis dígitos decimales de precisión. Los ceros no significativos no se visualizan. Si el número es muy grande o muy pequeño, se visualiza el número con un exponente prefijado por la letra E y dos dígitos (o tres dígitos si el tiepo es `double`). La anchura siempre es lo suficiente para mantener un signo menos y/o un exponente.

16.7. MANIPULADORES

La entrada y salida de datos realizada mediante los operadores de inserción y extracción puede formatearse ajustándola a izquierda o derecha, proporcionando una longitud mínima o máxima, precisión, etc.

La solución al problema son los *manipuladores* que son funciones especiales diseñadas específicamente para modificar el funcionamiento de un flujo. Los *manipuladores* manipulan el formato de un objeto. La biblioteca `iostream` viene con un número de manipuladores incorporados, aunque pueden

[1] La última edición (22.ª 2001) del diccionario de la Real Academia de la Lengua Española incorpora el término **formateado** como acepción informática.

añadirse otros fácilmente. La mayoría de las veces, los manipuladores se utilizan para indicar formateado, tal como la anchura de un campo, la precisión en números de coma flotante, etc. Sin embargo, los manipuladores no sólo pueden realizar formateado, sino otras tareas. Normalmente los manipuladores se utilizan en el centro de una secuencia de inserción o extracción de flujos. La mayoría de los manipuladores no tienen argumentos y están diseñados para realizar la tarea de formateado lo más sencilla posible.

Los manipuladores de flujos están incluidos, fundamentalmente, en el archivo de cabecera *iomanip*. Los manipuladores se pueden utilizar tanto para flujos de entrada como de salida. Consulte el manual de referencia de su compilador C++ para cualquier ampliación de la información de este capítulo.

La Tabla 16.4 recoge los manipuladores de flujo de E/S. Cualquiera de los manipuladores se puede insertar en la sentencia `cout` como cualquier otro elemento.

Tabla 16.4. Manipuladores de flujos.

Manipulador	Acción
`dec`	Utiliza conversión decimal (*por defecto*).
`hex`	Utiliza conversión hexadecimal.
`oct`	Utiliza conversión octal.
`ws`	Extrae caracteres espacios en blanco.
`endl`	Inserta nueva línea (se puede utilizar en lugar de `'\n'`).
`ends`	Añade un carácter terminal nulo al flujo de salida (`'\0'`).
`flush`	Limpia (fluye) un flujo de salida.
`setbase (n)`	Establece la base de conversión a *n* (0, 8, 10 o bien 16). 0 significa decimal por defecto.
`setprecision (n)`	Establece la precisión de coma flotante a *n*.
`setw (n)`	Establece la anchura del campo a *n*.
`setfill (c)`	Establece el carácter de relleno a *c*.
`setiosflags (f)`	Establece los bits de formato especificado por el argumento *f* de tipo `long`.
`resetiosflags (f)`	Pone a cero los bits de formato especificados por el argumento *f* de tipo `long`.

La sintaxis típica para utilizar los manipuladores es:

```
cout << setw (anchura del campo) << elemento de salida;
```

requiriendo la inclusión del archivo de cabecera *iomanip.h*

```
#include <iostream>
#include <iosmanip>
...
cout << setw(3) << i << setw(5) << i*i*i;
```

16.7.1. Bases de numeración

Normalmente, los enteros se visualizan como decimales (números escritos en base 10); es posible, sin embargo, seleccionar una base de numeración distinta (octal —8—, hexadecimal —16—) llamando a las funciones (manipuladores) `dec`, `hex` o bien `oct`. Así, por ejemplo,

```
cout << dec << Total << endl;
```

visualiza el valor de `Total` en base 10. `dec` es importante para volver a seleccionar la base 10 (decimal) después de haber trabajado con otras bases. Por ejemplo, si `Total` toma el valor decimal 255, la sentencia

```
cout << hex << Total << endl;
```

produce la salida

```
ff   valor hexadecimal correspondiente a 255 decimal
```

Se pueden entremezclar en una misma sentencia diversos manipuladores:

```
cout  << "Base 10 =" << dec << Total
      << "Base 16 =" << hex << Total << endl;
```

Ejemplo 16.14

```
// Uso de los manipuladores
#include <iostream>
using namespace std;

#include <iomanip>
using namespace std;

void main()
{
   cout << "Valor hex de" << 40 << "en decimal es:"
        << hex << 40 << endl;
   cout << "Valor octal de" << hex << 34 << "en hexadecimal es:"
        << oct << 34 << endl;
   cout << dec;    // Se restablece la numeración decimal
}
```

La salida del programa es:

```
Valor hex de 40 en decimal es: 28
Valor octal de 22 en hexadecimal es: 42
```

ya que `40` decimal equivale a `28 (2*16 + 8 = 40)` en hexadecimal, y `42` en base octal `(4*8 + 2 = 34)` equivale a `22` en hexadecimal `(2*16 + 2 = 34)`.

El manipulador `setbase(int n)` establece la base numérica a 8, 10 o 16. Este manipulador parametrizado funciona igual que los manipuladores 8, 10 o 16. Un ejemplo puede ser

```
cout << setbase(10);
cin >> setbase(10);
```

Ejemplo 16.15

```
#include <strstrea>
#include <iomanip>
```

```
void main()
{
   const int n = 100;

   // visualizar los resultados en distintas bases
   cout  << "\n" << n << " "
         << oct << n << " "
         << hex << n << endl;
}
```

La salida en pantalla es

```
100   144   64
```

16.7.2. Anchura de los campos

El manipulador `setw()` proporciona un medio para fijar la anchura del formato de salida. Por defecto, los datos que entran y salen se corresponden con el espacio necesario para almacenar ese dato, es decir, si se escribe una cadena de seis caracteres, la anchura de salida 6. El prototipo de `set` es

```
setw(int n)
```

Un ejemplo del uso de `setw()` para visualizar un campo de ocho caracteres de ancho es

```
cout << "12345678901234567890" << endl;
cout << setw(8) << "Hola";
cout << "Mundo";
```

que produce la salida siguiente:

```
12345678901234567890
   Hola mundo
```

Es decir, el ancho del campo `Hola` ha sido ocho caracteres y se alinea a derechas.

Ejemplo 16.16

```
// ANCHURA.CPP

#include <iostream>
using namespace std;

#include <iomanip>

void main()
{
   cout << setw(10) << "M" << setw(10) << "N" << endl;
   cout << setw(10) <<  1  << setw(10) << 7.77 << endl;
   cout << setw(10) <<  10 << setw(10) << 77.77 << endl;
   cout << setw(10) << 100 << setw(10) << 777.77 << endl;
}
```

La salida será

```
         M         N
         1      7.77
        10     77.77
       100    777.77
```

Se puede utilizar la función miembro `width()` para modificar la anchura del campo. El valor del parámetro pasado será la anchura en caracteres. Si se especifica una anchura en la entrada, lo que se hace es limitar la entrada a esa longitud. Así,

```
cout.width(5);
cout << "ABC" << "DEF" << "GHI"
```

visualiza

```
   ABCDEFGHI
```

y

```
cin.width(20);
```

limita la entrada a 20 caracteres.

16.7.3. Rellenado de caracteres

Siempre que se establece la anchura de un campo más grande que la anchura de los datos, los espacios adicionales se rellenan con caracteres blancos, que es el estado por defecto.

Se puede cambiar el carácter de relleno utilizando la función miembro `fill()`. Así, por ejemplo, supongamos que se desea que el carácter de relleno sea un asterisco (`*`); un código fuente que realiza esa tarea puede ser:

```
cout << "12345678901234567890" << endl;
cout.width(15);
cout.fill('*');
cout << "Hola Mackoy" << endl;
float Z  = 99.99;
cout.width(15);
cout << Z;
```

cuya salida será:

```
12345678901234567890
****Hola Mackoy
******99.989998
```

Obsérvese que el carácter de relleno permanece hasta que se vuelve a cambiar.

```
char Relleno;

Relleno = cout.fill('*');
```

```
cout << "Hola Mackoy" << endl;
...
cout.fill(Relleno); // Se recupera el antiguo carácter de relleno
```

16.7.4. Precisión de números reales

Si se visualiza un número en coma flotante, se visualizan hasta seis dígitos de precisión, por defecto. Los ceros a la derecha del punto decimal se suprimen. Se puede cambiar el número de dígitos de precisión de seis a otro valor con el manipulador `setprecision()`. Su argumento entero (*n*) especifica el número de dígitos significativos que se visualizarán. El formato es:

```
cout << setprecision(int i);
```

Ejemplo 16.17

```
// Archivo PRECISIO.CPP
// Fija el número de posiciones decimales

#include <iostream>
using namespace std;

#include <iomanip>

void main()
{
   float prueba = 814.159265;
   cout  << setprecision(2)   // 2 dígitos significativos
         << prueba << endl;
   cout  << setprecision(3)   //3 dígitos significativos
         << prueba << endl;
   cout  << setprecision(4)   //4 dígitos significativos
         << prueba << endl;
   cout  << setprecision(5)   //5 dígitos significativos
         << prueba << endl;
}
```

La salida del programa es:

```
814.16
814.159
814.1592
814.15924
```

Se puede utilizar también la función miembro **`precision()`** para establecer la precisión. Por ejemplo, la sentencia fija la precisión a 3, para la salida correspondiente:

```
cout.precision(3);
```

16.8. INDICADORES DE FORMATO

Cada flujo, es decir, cada objeto de la clase istream y ostream contiene un conjunto de informaciones (*indicadores*) que especifican cúal es en un momento dado su *«estatuto de formato»*. Este modo de pro-

ceder es muy diferente del empleado por las funciones de C, tales como **`printf`** o **`scanf`**, en las que a cada operación de entrada/salida se le proporcionan los indicadores de formateado apropiado.

El método empleado por C++ es más eficiente que el empleado por C, ya que permite, eventualmente, al usuario ignorar totalmente el método empleado por C, un tanto complejo por otra parte.

Cada uno de estos indicadores (*flags*) pueden establecerse o reinicializarse utilizando un manipulador incorporado. Los manipuladores `setiosflags(long)` y `resetiosflags(long)` realizan fundamentalmente estas tareas.

16.8.1. Uso de `setiosflags()` y `resetiosflags()`

El manipulador `setiosflags()` se define como

```
setiosflags(long f)
```

y sirve para especificar, por ejemplo, si un dato se alinea a izquierda o derecha en un campo. Por defecto, los valores se alinean a derecha en un campo. La siguiente sentencia activa la opción de alineación a izquierda:

```
cout << setiosflags(ios::left);
```

Los argumentos de los manipuladores (bits indicadores de `ios`) realizan diversas tareas que se recogen en la Tabla 16.5.

Tabla 16.5. Bits de estado (palabra de estado del formateado).

Bit de estado (argumento)	Propósito
`skipws`	Salta espacios en blanco en operaciones de entrada.
`left`	Justifica la salida a la izquierda del campo.
`right`	Justifica la salida a la derecha del campo.
`internal`	Rellena el campo después del signo o el indicador base.
`dec`	Activa conversión decimal.
`oct`	Activa conversión octal.
`hex`	Activa conversión hexadecimal.
`showbase`	Visualiza el indicador de base numérica. *Ejemplo:* `044 (número octal)` `0x2ea7 (número hex)`
`showpoint`	Visualiza punto decimal en valores de coma flotante. *Ejemplo:* `456.00`
`uppercase`	Visualiza valores hexadecimales en mayúsculas. *Ejemplo:* `4BFE`
`showpos`	Visualiza números enteros positivos precedidos del signo +.
`scientific`	Notación científica en los números en coma flotante. *Ejemplo:* `3.1416 e + 00`
`fixed`	Utiliza notación en coma fija para números en coma flotante. *Ejemplo:* `1234.5`
`unitbuf`	Vacía (limpia) las memorias (*buffers*) después de cada escritura.
`stdio`	Vacía (limpia) las memorias intermedias después de cada escritura sobre `stdout o stderr`.

Se debe hacer preceder a cada argumento o bit de estado de la cláusula `ios::`, que significará su asociación con la clase `ios`. Por ejemplo,

```
float pi = 3.14159;
cout << setiosflags(ios::fixed) << pi << endl;
```

ha seleccionado un valor de coma flotante con notación fija y la salida será:

```
3.14159
```

Se selecciona la notación científica utilizando la sentencia siguiente:

```
cout << setiosflags(ios::scientific) << pi << endl;
```

y se visualiza la salida siguiente:

```
3.14159e+00
```

Se pueden establecer múltiples indicadores en una operación mediante operadores `OR`. Por ejemplo,

```
cout << setiosflags(ios::dec|ios::showbase) << Total << endl;
```

Para limpiar o reponer los indicadores de estado, se deben utilizar **`resetiosflags()`**. Por ejemplo, para limpiar o borrar el parámetro `showbase`, escribir

```
cout << resetiosflags(ios::showbase) << Total << endl;
```

Así por ejemplo, si se activa la opción de alineación a izquierda

```
cout << setiosflags(ios::left);
```

se desactivará con la sentencia

```
cout << resetiosflags(ios::left);
```

Otro ejemplo para visualizar resultados en diferentes bases de numeración son las siguientes líneas de código:

```
cout  << setiosflags(ios::showbase)
      << "\n" << v << " "
      << oct << v << " "
      << hex << v << endl;
```

que produce la salida:

```
100   0144   0x64
```

Ejemplo 16.18

El programa `FORMATO1.CPP` muestra un sistema para formatear datos de salida de diversas formas.

```
// Archivo FORMATO1.CPP
```

```
#include <iostream>
using namespace std;

#include <iomanip>

void main()
{
   float v1 = 4500.25;
   float v2 = 325.99;
   float v3 = 54225.00;

   cout  << setiosflags(ios::showpoint|ios::fixed)
         << setprecision(2)
         << setfill('*')
         << setiosflags(ios::right);
   cout  << "\n Saldo Final: $" << setw(10) << v1 << endl;
   cout  << "\n Saldo Final: $" << setw(10) << v2 << endl;
   cout  << "\n Saldo Final: $" << setw(10) << v3 << endl;
}
```

Al ejecutar el programa se produce:

```
Saldo Final: $***4500.25
Saldo Final: $****325.99
Saldo Final: $**54225.00
```

Ejemplo 16.19

```
// Archivo FORMATOS2.CPP

#include <iostream>
using namespace std;

#include <iomanip>

void main()
{
   const float p = 3.14159;
   // Visualizar números reales con diversos manipuladores
   cout  << setiosflags(ios::showpos|ios::scientific)
         << "\n El valor de PI es"
         << setprecision(3)
         << setw(15) << setfill('*')
         << setiosflags(ios::right)
         << p;
}
```

La salida de este programa es

```
El valor de PI es*****+3.142e+00
```

16.8.2. Las funciones miembro `setf()` y `unsetf()`

Existe un segundo método para establecer los indicadores de flujo: llamar a las funciones miembro **`setf()`** y **`unsetf()`**. Estas funciones son similares a los manipuladores `setiosflags()` y `resetiosflags()`. La diferencia reside en que **`setf()`** y **`unsetf()`** son verdaderas funciones miembro. Se accede a las funciones miembro directamente:

```
cout.setf(ios::scientific);
cout.unsetf(ios::scientific);
```

16.9. ARCHIVOS C++

C++ utiliza flujos (*streams*) para gestionar flujos de datos, incluyendo el flujo de entrada y de salida. Un **archivo** es una secuencia de bits almacenados en algún dispositivo externo tal como un disco o una cinta magnética. Los bits se interpretan de acuerdo al protocolo de algún sistema software. Si estos bits se agrupan en bytes de 8 bits interpretados por el código ASCII, entonces el archivo se denomina *archivo de texto* y puede ser procesado por editores estándar. Si los bits se agrupan en palabras de 32 bits representando a pixeles de colores, entonces el archivo es un *archivo gráfico* y se procesa por un software de gráfico especializado. Si el archivo es un programa ejecutable, entonces sus bits se interpretan como instrucciones al procesador de la computadora.

En C++, un archivo es simplemente un flujo externo: una secuencia de bytes almacenados en disco. Si el archivo se abre para salida, es un flujo de archivo de salida. Si el archivo se abre para entrada, es un flujo de archivo de entrada.

La biblioteca de flujos contiene tres clases, `ifstream`, `ofstream` y `fstream`, y métodos asociados para crear archivos y manejo de entrada y salida de archivos.

Las tres clases, `ifstream`, `ofstream` y `fstream`, se declaran en el archivo de cabecera *fstream*, que, incidentalmente, incluye el archivo de cabecera *iostream* de modo que no necesita definir explícitamente `#include <iostream` si se incluye el archivo de cabecera *fstream* (`#include <fstream>`).

C++ soporta dos tipos de archivos: texto y binario. Los *archivos de texto* almacenan datos como códigos ASCII. Los valores simples, tales como números y caracteres únicos, están separados por espacios. Los archivos de texto se pueden utilizar para almacenamiento de datos o crear imágenes de salida impresa que se pueden enviar más tarde a una impresora.

Los *archivos binarios* almacenan flujos de bits, sin prestar atención a los códigos ASCII o a la separación de espacios. Son adecuados para almacenar objetos. Sin embargo, el uso de los archivos binarios requiere usar la dirección de una posición de almacenamiento.

16.10. APERTURA DE ARCHIVOS

Antes de que un programa pueda leer o escribir de un disco, se debe *abrir* el archivo. El proceso de la apertura de un archivo identifica la posición del archivo en el programa. Para abrir un archivo de texto C++ para lectura, se crea un objeto (un flujo) de la clase `ifstream`; para abrir un archivo para escritura, se crea un objeto de la clase `ofstream`. Se puede entonces utilizar los nombres de los flujos que se crean con los operadores de inserción y extracción.

Al igual que los flujos, por defecto, `cin y cout`, los flujos de E/S de archivos ANSI pueden transferir datos sólo en una dirección. Esto significa que se deben abrir y manipular flujos independientes para lectura y escritura de archivos de texto. La biblioteca de flujos C++ ofrece un conjunto de funciones miembro comunes a todas las operaciones de E/S de flujo de archivo.

Para *abrir el archivo para lectura* al comienzo de cada ejecución de programa, el programa incluye la sentencia

```
ifstream aen ("demo");
```

El objeto `ifstream` se llama `aen`. Está asociado con un archivo cuyo nombre del camino está dentro de la lista de parámetros. Este parámetro se pasa al constructor de clase, que se utiliza para localizar y abrir el archivo.

La *apertura de un archivo para escritura o salida* se realiza con la sentencia `ofstream` definiendo un objeto de la clase `ofstream` (*output file stream*).

```
ofstream archsal ("copy.out", ios_base::out);
```

Un archivo `ofstream` se puede abrir de dos maneras: (1) en modo salida `(ios_base::out)`; (2) en modo añadir `(ios_base::app)`. Por defecto, un archivo `ofstream` se abre en modo salida. La definición de `archsa1` es equivalente a la del archivo de salida.

```
// abierta en modo salida por defecto
ofstream archsalida("copy.sal");
```

Si un archivo existe se abre en modo salida, todos los datos almacenados en ese archivo se descartan. Si se desea *añadir* en lugar de *reemplazar* los datos dentro de un archivo existente, se debe abrir el archivo en modo *append* («añadir»). Los datos adicionales escritos en el archivo se añaden a continuación de su extremo final. *De cualquier manera, si el archivo no existe, se creará.*

Precaución

Antes de intentar leer o escribir en un archivo, es siempre una buena idea verificar que se ha abierto con éxito. Se puede comprobar `archsal` del modo siguiente:

```
if (!archsal)  {  // apertura fallida
   cerr << "no se puede abrir copy.sal para salida\n";
   exit (-1);
}
```

Ejercicio 16.1

El siguiente programa obtiene caracteres de la entrada estándar y los pone en el archivo `copy.sal.`

```
#include <fstream>

int main()
{
   // abrir un archivo copi.sal para salida;
   ofstream archsal ("copi.sal");

   if (!archsal) {
     cerr << "No se puede abrir copi.sal para salida:" << endl;
        return –1;
   }
   char car;
   while (cin.get (car))
     archsal.put (ch);
   return 0;
}
```

16.10.1. Apertura de un archivo sólo para entrada

Para esta operación se utiliza un objeto de `ifstream`. La clase `ifstream` se deriva de `istream`.

Ejercicio 16.2

El siguiente programa lee un archivo especificado por el usuario y escribe su contenido en la salida estándar.

```
#include <fstream>
using namespace std;

int main()
{
   cout << "nombre archivo:";
   char nombre_arch[80];

   cin >> nombre_arch;

   // abrir un archivo copy.sal para entrada
   ifstream ArchEn (nombre_arch.));

   if (!ArchEn)  {
      cerr << "incapaz abrir archivo de entrada:"
            <<nombre_arch << "precaución de salida\n";
      return -1;
   }

   char car;
   while (ArchEn.get(car))
          cout.put (car);
   return 0;
}
```

Un archivo se puede desconectar del programa invocando la función miembro `close()`. Por ejemplo,

```
ArchivoAct.close();
```

16.11. E/S EN ARCHIVOS

El sistema de E/S de C++ se puede utilizar para hacer E/S a archivos. Para poder realizar las operaciones de E/S, se necesita incluir el archivo de cabecera *fstream* en el que se definen varias clases con diferentes funciones miembro. El archivo de cabecera `<fstream>` declara las clases `ifstream, ofstream` y `fstream`.

Específicamente la clase `ifstream` se deriva de la clase `istream` y permite a los usuarios acceder a archivos y leer datos de ellos. La clase `ofstream` se deriva de la clase `ostream` y permite a los usuarios acceder a archivos y escribir datos en ellos. Por último, la clase `fstream` se deriva de las clases `ifstream y ofstream` y permite a los usuarios acceder a los archivos para entrada y salida de datos. Para abrir y gestionar adecuadamente las clases `ifstream y ofstream` relacionadas con un sistema de archivos, se debe declarar con un constructor apropiado.

```
ifstream();
ifstream (const char*, int = ios::in, int prot = filebuf::openprot);
```

```
ofstream();
ofstream (const char*, int = ios::out, int prot = filebuf::openprot);
```

El constructor sin argumentos crea una variable que se asociará posteriormente con un archivo de entrada. El constructor de tres argumentos toma como su primer argumento el archivo con nombre. El segundo argumento especifica el modo archivo. El tercer argumento es para protección de archivos.

- La apertura de un flujo de entrada se debe declarar en la clase `ifstream`.
- La apertura de un flujo de salida se debe declarar en la clase `ofstream`.
- Los flujos que realicen operaciones de entrada y salida deben declararse en la clase `fstream`.

Ejemplo

```
ifstream ent;      // crea un flujo de entrada
ofstream sal;      // crea un flujo de salida
fstream ambos;     // crea un flujo de entrada y salida
```

16.11.1. La función `open`

Una vez creado el flujo, se puede utilizar la función `open()` para asociarlo con un archivo. Esta función es miembro de las tres clases de flujo. La declaración de la función `open()` (su prototipo) tiene por prototipos:

```
// abre archivos ifstream
void open (const char*, int = ios::in,
           int prot = filebuf::openprot);

// abre archivo ofstream
void open (const char*, int = ios::out,
           int prot = filebuf::openprot);
void close();
```

Estas funciones se pueden utilizar para abrir y cerrar archivos apropiados. El prototipo estándar es:

```
void open(const char*nomarch, int modo, int acceso=filebuf::openprot);
```

nombre del archivo (puede incluir un especificador de vía de acceso)

determina cómo se abrirá el archivo

protección del archivo

Los posibles valores del argumento *modo* se definen como enumeradores de la clase `ios` (Tabla 16.6).

Tabla 16.6. Valores del argumento modo de archivo.

Argumento	**Modo**
`ios::in`	Modo entrada
`ios::app`	Modo añadir
`ios::out`	Modo salida
`ios::ate`	Abrir y buscar el fin del archivo
`ios::nocreate`	Genera un error si no existe el archivo
`ios::trunc`	Trunca el archivo a 0 si existe ya
`ios::noreplace`	Genera un error si el archivo existe ya
`ios::binary`	El archivo se abre en modo binario

El argumento *acceso* especifica el permiso de acceso por defecto que se asigna a un archivo si se crea por la función constructor. El valor por defecto es `filebuf::openprot`.

Ejemplos

```
// Ejemplo 1. Abre el archivo AUTOEXEC.BAT para entrada de texto
char* cAutoExec = "\\AUTOEXEC.BAT";
fstream f;
// abrir para entrada
f.open (cAutoExec, ios::in);

// Ejemplo 2. Abre el archivo DEMO.DAT para salida de texto
fstream f;
// abrir para salida
f.open ("DEMO.DAT", ios::out);

// Ejemplo 3. Abre el archivo PRUEBAS.DAT para entrada y
// salida binaria
fstream f;
// abrir para E/S de acceso aleatorio
f.open ("PRUEBAS.DAT", ios::in | ios::out | ios::binary);
```

Consejo

Para abrir un flujo para entrada y salida, se deben especificar los dos valores del argumento *modo*, `ios::in` e `ios::out`, como se muestra en el siguiente ejemplo:

```
fstream miflujo;
miflujo.open ("prueba", ios::in |ios::out);
```

Si `open()` falla, el flujo será nulo. A pesar de que es sintácticamente válido abrir un archivo mediante la función `open()`, no se suele hacer, ya que las clases `ifstream`, `ofstream` y `fstream` disponen de funciones constructor que abren automáticamente el archivo. Las funciones constructor tienen los mismos parámetros y valores por omisión que la función `open()`. En consecuencia, la forma habitual de abrir un archivo será la siguiente:

```
ifstream miflujo ("miarch"); // abre el archivo para entrada
```

Si no se puede abrir el archivo por algún motivo, el valor de la variable de flujo asociada `miflujo` será cero. Se puede utilizar el siguiente código para confirmar que el archivo se ha abierto correctamente:

```
ifstream miflujo ("archdemo");
if (!miflujo) {
   cout << "No se puede abrir el archivo\n";    // error
}
```

Ejercicio 16.3

El siguiente ejemplo muestra el uso de las clases `ifstream` y `ofstream`. En el ejemplo, un archivo denominado fuente se abre para lectura y otro archivo que se denomina destino se abre para escritura. Si ambos archivos se abren con éxito, el contenido del archivo fuente se copia en el archivo destino. Si un archivo no se puede abrir con éxito, se señala un error:

```
ifstream fuenteF("fuente");
ofstream destinoF("destino");
if(fuenteF || destinoF)
   cerr << "Error fuente o destino open failed\n";
else
   for(char c = 0; destino && fuente.get(c);)
      destino.put(c);
```

16.11.2. La función `close`

La función miembro `close` cierra el archivo al que se conecta el objeto de flujo. La sintaxis de `close` es:

```
void close();
```

Ejemplo

```
char cAutoExec = "AUTOEXEC.BAT";
fstream f;
// abrir para entrada
f.open(cAutoExec, ios::in);
// sentencias de E/S
f.close()   // cerrar bufer del flujo del archivo
```

Este ejemplo abre el archivo `AUTOEXEC.BAT` para entrada, realiza operaciones de entrada y, a continuación, cierra el búfer del flujo del archivo.

> La función `close ()` no utiliza parámetros ni devuelve ningún valor.

Otras funciones miembro

Además de las funciones miembro `open` y `close`, la biblioteca de flujos C++ ofrece las siguientes funciones miembro y operadores:

- La función miembro **`good`**, que devuelve un valor distinto de cero si no existe ningún error en una operación de flujo. La declaración de esta función miembro es:

  ```
  int good();
  ```

- La función miembro **`fail`**, que devuelve un valor distinto de cero si existe un error en una operación de flujo. La declaración de esta función miembro es:

  ```
  int fail();
  ```

- La función miembro **`eof`**, que devuelve un valor distinto de cero si el flujo ha alcanzado el final del archivo. La declaración de esta función miembro es:

  ```
  int eof();
  ```

- El operador sobrecargado **`!`**, que determina el estado del error. Este operador toma un objeto de flujo como un argumento.

Además de las funciones miembro heredadas de las clases `iostream`. Las clases `ifstream`, `ofstream` y `fstream` definen también sus propias funciones específicas:

`void attach(int fd)`	Conecta el objeto flujo al flujo referenciado por el descriptor del archivo `fd`.
`filebuf* rdbuf()`	Devuelve un array `filebuf` asociado con el objeto flujo.

16.12. LECTURA Y ESCRITURA DE ARCHIVOS DE TEXTO

La lectura y escritura en un archivo de texto se puede realizar con los operadores << y >> con el flujo abierto.

Ejercicio 16.4

El siguiente programa escribe un entero y un valor en coma flotante y una cadena en un archivo llamado DEMO.

```
#include <iostream>
using namespace std;

#include <fstream>
int main()
{
   ofstream sal ("demo");
   if (!sal) {
     cout << "No se puede abrir el archivo" << endl;
     return 1;
   }

   sal << 10 << " " << 325.45 << endl;
   sal << "Ejemplo de archivo de texto" << endl;

   sal.close();

   return 0;
}
```

Ejercicio 16.5

El programa siguiente lee un número entero, un número en coma flotante, un carácter y una cadena del archivo `DEMO`.

```
#include <iostream>
#include <fstream>

int main()
{
   char c;
   int i;
```

```
    float f;
    char cad[40];

    ifstream ent ("demo");
    if (!ent)
    {
       cout << "No se puede abrir el archivo" << endl;
       return 1;
    }
    ent >> i;
    ent >> f;
    ent >> c;
    ent >> cad;

    ent.close();
    return 0;
}
```

El operador >> produce en la lectura de archivos de texto una conversión de ciertos caracteres. Por ejemplo, se omiten los caracteres en blanco.

Ejercicio 16.6

Escribir datos en un archivo de datos externo.

El archivo de cabecera `<fstream>` define la clase `ofstream` que debe ser *instanciada* para escribir en un archivo externo.

Es preciso abrir un archivo `notas.dat` como archivo de salida, construir el flujo de salida `archsal` y conectar ese flujo al archivo. Estas operaciones se realizan invocando al constructor de la clase `ofstream`.

El algoritmo de escritura en un archivo de datos externo prosigue asegurando que el archivo se ha abierto adecuadamente. Para ello se invoca al operador de negación sobrecargado (`!archsal`) y en caso de que se produzca un error, se visualiza un mensaje de error y se termina el programa.

```
cerr << "Error: no se puede abrir archivo de Salida" << endl;
exit(1)
```

Si el proceso es correcto, el programa utiliza un bucle de entrada para leer nombres, identificar los números de expediente del alumno (`exp`) y las calificaciones (`notas`) de la entrada estándar y escribirlas en un archivo externo.

```
#include <iostream>    // define el flujo cout
#include <fstream>     // define la clase ofstream
using namespace std;

#include <iostream>
#include <stdlib.h>      // define la función exit()
using namespace std;
```

```
main()
{
   ofstream archsal("notas.dat", ios::out);
   if (!archsal)  {
     cerr << "Error: no se puede abrir archivo de salida" << endl;
     exit(1);
   }
   char exp[6], nombre[20];
   int nota;
   cout << "\t1:";
   int n = 1;
   while (cin >> id >> nombre >> nota)  {
      archsal << nombre << " " << exp << " " << nota << endl;
      cout << "\t" << ++n << ":";
   }
   archsal.close();
}
```

Al ejecutar el programa se introducen los datos de alumnos siguientes:

```
1: Mackoy         951234  7
2: Carrigan       962146  8
3: Mortimer       991045  9
4: Mackena        981146  9
5: García         982045  5
6: Rodríguez      991122  1
7: López          961333  6
```

El archivo creado `notas.dat` es

```
Mackoy     951234  7
Carrigan   962146  8
Mortimer   991045  9
Mackena    981146  9
García     982045  5
Rodríguez  991122  1
López      961333  6
```

Ejercicio 16.7

Leer datos de un archivo de datos externo `(notas.dat)`.

La clase `iostream` se define en el archivo de cabecera `<fstream>`. Esta clase debe ser instanciada para leer de un archivo externo.

```
#include <iostream>    // define el flujo cout
#include <fstream>     // define la clase ofstream
#include <stdlib>      // define la función exit()
main()
{
   ifstream archen ("notas.dat, ios::in);
```

```
   if (!archen) {   // archivo de entrada, archen
       cerr << "Error: no se puede abrir archivo de entrada" << endl;
       exit(1);
   }
   char exp[6], nombre[20];
   int nota, suma = 0, cuenta = 0;
   while (archen >> exp >> nombre >> nota) {
     suma += nota;
     ++cuenta;
   }
   archen.close();
   cout << "La nota media es" << float(suma)/cuenta << endl;
}
```

Si se ejecuta el programa anterior la salida será:

```
La nota media es 6.42857
```

16.13. E/S BINARIA

Existen varios modos de escribir y leer datos en binario desde un archivo. Un método es utilizar las funciones miembro `put()` y `get()`. Otro método es utilizar las funciones miembro `read()` y `write()`.

16.13.1. Funciones miembro `get` y `put`

La función miembro **`get()`** de la clase `istream` lee el flujo de entrada byte a byte (carácter). Existen tres formatos de la función `get()`:

1. `istream & get (char& car);`

   Extrae un único carácter del flujo de entrada, incluyendo espacio en blanco y lo almacena en `car`. Devuelve el objeto `istream` al que se aplica.

Ejemplo 16.20

El siguiente programa muestra en pantalla el contenido de un archivo.

```
#include <iostream>
#include <fstream>

int main(int argc, char *argv[])
{
   char c;

   if (argc != 2) {
     cout << "error en número de archivos\n";
     return 1;
   }
```

```
    ifstream ent(argv[1], ios::in | ios::binary);
    if (!ent)
    {
      cout << "No se puede abrir el archivo" << endl;
      return 1;
    }

    while (ent)      // ent será 0 cuando se alcance eof
    {
      ent.get(c);
      cout << c;
    }

    ent.close();

    return 0;
}
```

El bucle `while` se termina cuando `ent` alcanza el final del archivo (`eof`, *end of file*) cuyo valor será nulo.

Consejo

Existe un modo más abreviado de implementar el bucle de lectura, aunque menos legible.

```
while (ent.get(c))
   cout << c;
```

16.13.2. Función `put()`

La función miembro `put()` proporciona un método alternativo de sacar (escribir) un carácter en el flujo de salida. `put()` acepta un argumento de tipo `char` y devuelve el objeto de la clase `ostream` a la cual se invoca.

Sintaxis

```
ostream& put(char&c);
```

Ejemplo 16.21

Escribir una cadena de caracteres no ASCII en el archivo.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
```

```
    char *p = "Hola colegas \n\r\xff";
    ofstream sal("prueba", ios::out | ios::binary);
    if (!sal)
    {
       cout << "No se puede abrir el archivo" << endl;
       return 1;
    }

    while (*p) sal.put(*p++);

    sal.close();

    return 0;
 }
```

16.13.3. Formato 2 de `get()`

El segundo formato de `get()` lee también un único carácter del flujo de entrada. La diferencia es que devuelve el valor en lugar del objeto `istream` al que se aplica. Devuelve un tipo `int` en lugar de `char` debido a que devuelve también la representación final de archivo (**`eof`**), que se suele representar como –1 para diferencia de los valores del conjunto de caracteres.

Se ha de comprobar si el valor devuelto es final de archivo, para lo cual se compara con la constante `EOF` definida en el archivo de cabecera `iostream`. La variable asignada para contener el valor devuelto por `get()` se debe declarar del tipo `int`, de modo que pueda contener valores carácter y `EOF`.

Ejemplo

```
#include <iostream>
using namespace std;

int main()
{
   int car;
   while ((car = cin.get()) != EOF)
     cout.put(car);

   return 0;
}
```

16.13.4. Funciones `read` y `write`

El segundo método de lectura y escritura de datos binarios consiste en utilizar las funciones miembro `read()` y `write()`.

Sintaxis de `write()`

```
ostream& write(const char* buf, int num);
ostream& write(const unsigned char* buf, int num);
ostream& write(const signed char* buf, int num);
```

La función `write()` inserta (escribe) el número especificado de bytes (*num*) en el flujo asociado desde el búfer al que apunta *buf* (*buf* es un puntero a un array de caracteres y *num* es el número de caracteres a escribir).

Sintaxis de `read()`

```
istream &read(char* buf, int num);
istream &read(unsigned char* buf, int num);
istream &read(signed char* buf, int num);
```

La función `read()` lee *num* bytes del flujo asociado y los coloca en el búfer al que apunta *buf*. El parámetro es un puntero a un array de caracteres y *num* especifica el máximo número de caracteres a leer. La función miembro extrae caracteres hasta que se alcanza el final del archivo.

Ejemplo 16.22

Escritura de un array de enteros y lectura posterior.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   int n[6] = {15, 25, 35, 45, 50, 60};
   register int i;
   ofstream salida ("prueba", ios::out | ios::binary);
   if(!salida)
   {
     cout << "No se puede abrir el archivo" << endl;
     return 1;
   }

   salida.write ((unsigned char*) n, sizeof) n));

   salida.close();

   // inicializa el array
   for (i = 0; i < 6; i++)
        n[i] = 0;

   ifstream entrada ("prueba", ios::in | ios::binary);
   entrada.read ((unsigned char*) n, sizeof) n));

   // presenta valores leídos del archivo
   for (i = 0; i < 6; i++)
      cout << n[i] << " ";

   entrada.close();
   return 0;
}
```

Caracteres leídos

Si se alcanza el final del archivo antes de que se haya leído el número de caracteres especificado, la función `read()` se interrumpe, y el búfer contiene tantos caracteres como están disponibles. Si se desea conocer el número de caracteres que se han leído, se puede utilizar la función miembro `gcount()`, que devuelve el número de caracteres leídos en la última operación de entrada binaria.

Sintaxis

```
int gcount();
```

Detección de EOF

El final del archivo se puede detectar mediante la función miembro `eof()`, que devuelve un valor distinto de cero cuando se alcanza el final del archivo; en caso contrario, devuelve 0.

Sintaxis

```
int eof();
```

16.14. ACCESO ALEATORIO

En el sistema de E/S de C++ se pueden realizar accesos aleatorios mediante las funciones `seekg()` y `seekp()`. Este tipo de acceso es usual en trabajos de bases de datos que permiten seleccionar y elegir registros especificados en archivos. Estas tareas se pueden realizar con las dos funciones miembro sobrecargadas heredadas de la clase `istream`.

El acceso a archivos aleatorios se describe generalmente en términos de un *puntero de archivo* (no confundir con punteros C++). Un puntero de archivo es un índice o apuntador que apunta a una posición dada o a la *posición actual* de un archivo entre el principio y el final de éste. La posición actual es el punto en el que comienza el siguiente acceso al archivo. Dado que el acceso a archivo se puede clasificar en términos de entrada y salida, la posición actual se puede referenciar como posición actual «obtener» (*current get position*) y una posición actual de «poner» (*current put position*). Las posiciones actuales *get* y *put* ayudan a introducir las funciones miembro asociadas de las clases `istream y ostream` para realizar acceso aleatorio o no secuencial.

El sistema de E/S de C++ maneja dos punteros asociados con el archivo. Uno es el puntero *get*, que especifica la posición del archivo en el que se producirá la siguiente operación de entrada. El otro es el puntero *put*, que especifica la posición del archivo en el que se producirá la siguiente operación de salida. Cada vez que se realiza una operación de entrada o salida, el puntero correspondiente se avanza automáticamente. Sin embargo, cuando se utilizan las funciones `seekg()` y `seekp()` se puede acceder al archivo de modo aleatorio, no secuencial.

Las funciones de acceso aleatorio a archivos `seekg() y tellg()` tienen la sintaxis siguiente:

```
istream& seekg (streampos pos);
istream& seekg (streamoff desp, seek_dir dir);
streampos tellg();
```

La versión de un argumento de `seekg()` mueve la posición actual *get* a una posición absoluta *pos*, en el flujo de entrada. El argumento *pos* es de tipo `streampos`, que es, simplemente, un `long typedef` definido en el archivo de cabecera *iostream*.

```
typedef long streampos;
```

La versión de dos argumentos de `seekg()` mueve *desp* bytes relativos a la posición actual *get* en la dirección especificada por *dir*. Al igual que `streampos`, `streamoff` está definido en el archivo *iostream*.

```
typedef long streamoff;
```

El tipo de `dir` es un tipo enumerado `seek_dir` y está declarado como miembro de la clase `ios`:

```
enum seek_dir
{
   beg,         // principio del archivo
   cur,         // posición actual
   end,         // final del archivo
};
```

La función `seekg()` desplaza el puntero *get* del archivo asociado un número de bytes especificado en el desplazamiento *desp*, desde el origen *dir* que tomará uno de los tres valores del tipo `enum seek_dir`.

```
ios::beg
ios::cur
ios::end
```

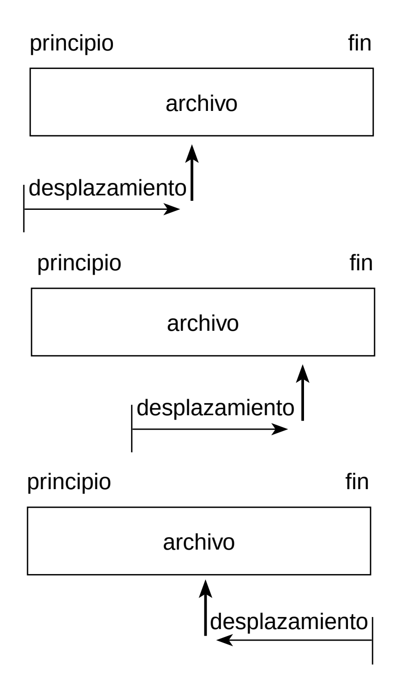


Figura 16.6. Posiciones de desplazamiento.

La función miembro `tellg()` devuelve la posición actual del flujo de entrada y devuelve un valor de –1 si se produce un error

```
streampos tellg();
```

`streampos` es un tipo definido en *iostream* `(typedef long)` capaz de almacenar el mayor valor que ambas funciones pueden devolver.

Ejemplo

```
// uso de las funciones miembro seekg() y tellg()
// de la clase istream para extraer la posición actual get
#include <fstream>        // E/S de archivo
using namespace std;

void main()
{
   char nombre_arch[] = "CGP.CPP";
   ifstream arch_en (nombre_arch);
   if (!arch_en)
      cerr << "no se puede abrir el archivo\""
           << nombre_arch << "\" " << endl;

   // determinar la longitud del archivo
   streampos inicio = arch_en.seekg(0, ios::beg).tellg();
   streampos fin    = arch_en.seekg(0, ios::end).tellg();

   // obtener un desplazamiento
   streamoff desplazamiento;
   cout << "introducir un desplazamiento (+) desde el principio
           del archivo |""
        << nombre_arch << "\"" << endl
        << cout cuya longitud tenga una longitud de archivo"
        << "(fin-inicio) << "caracteres:";
   cin  >> desplazamiento;

   // ir a desplazamiento
   arch_en.seekg (desplazamiento, ios::beg);

   // visualizar archivo en la pantalla
   char car;

   while (arch_en)
   {
     arch_en.get(car);
     cout << car;
   }
   // cerrar
   arch_en.close();
}
```

De igual modo, la clase `ostream` tiene dos funciones miembro: **`seekp()`** y **`tellp()`.** Su sintaxis es:

```
ostream& seekp(streampos pos);
ostream& seekp(streamoff desp, seek_dir dir);
streampos tellp();
```

La versión de un argumento de `seekp()` mueve la posición actual *put* a una posición absoluta, `pos`, en el flujo de salida. La versión de dos argumentos de `seekp()` mueve *desp* bytes relativos a la posición *put* actual en la dirección especificada por *dir*. La función miembro `tellp()` devuelve la posición actual del flujo de salida y devuelve un valor de –1 si ocurre un error.

Ejercicio 16.8

El siguiente programa permite especificar un nombre de archivo en la línea de órdenes, seguido del número del byte que se desea modificar en el archivo. Luego se escribe una 'T' en la posición especificada.

```
#include <iostream>
#include <fstream>
#include <stdlib>
using namespace std;

int main(int argc, char *argv[])
{
   if(argc != 3) {
      cout << "Usar: CAMBIA <nombre_archivo> <byte>\n";
      return 1;
   }
   fstream sal(argv[1], ios::in | ios::out | ios::binary);
   if (!sal) {
      cout << "No se puede abrir el archivo\n";
      return 1;
   }

   sal.seekp(atoi(argv[2]), ios::beg);

   sal.put('T');
   sal.close();

   return 0;
}
```

RESUMEN

En este capítulo ha aprendido a manejar las operaciones básicas de entrada y salida. Asimismo, ha aprendido a utilizar los manipuladores para ajustar opciones de formatos para seleccionar entradas y salidas. Las características típicas de E/S son:

- Cuatro objetos de flujos se crean en cada programa: `cout, cin, cerr` y `clog`.
- La salida a pantalla utiliza el identificador de flujo `cout` y el operador de inserción.

```
cout << valor;
```

- La entrada del teclado se realiza mediante el identificador de flujo `cin` y el operador de inserción >>.

```
cin >> variable;
```

- La función miembro **`get()`** se utiliza para leer caracteres individuales desde el teclado y se puede utilizar **`put()`** para sacar caracteres.
- Cuando se necesita leer una línea completa de entrada, se debe utilizar la función miembro `getline()`.
- Los manipuladores proporcionan un medio para modificar el flujo de entrada o salida. Un ejemplo típico, muy utilizado, es `endl` que sirve para insertar una nueva línea o retorno de carro.

```
cout << setw(int i)
cout << hex
cout << oct
cout << setprecision(int i)
cout << setiosflags(long indicador)
cout << resetiosflags(long indicador)
```

- Algunas funciones miembros utilizadas en el capítulo son:

```
cout.write(cadena, int i)
cout.put(char car);
cout.get([char c]);
cin.get(cadena, int i[, char term]);
cin.getline(cadena, int [, char term]);
```

Un archivo de datos es una secuencia de bits almacenados en algún dispositivo de almacenamiento externo (cinta, disco, disquete, CD-ROM). Excepto en circunstancias extremas, los archivos externos son permanentes: se pueden crear y guardar un día y recuperarlos posteriormente del disco en su computadora.

Se puede crear un archivo de datos externo utilizando un editor de texto de igual forma que se crea un archivo de programa y también crearse durante la ejecución de un programa. Un archivo de datos se puede leer muchas veces. De igual forma se puede instruir a un programa para que ejecute su salida en un archivo de disco en vez de visualizarlo en la pantalla de su monitor.

La escritura de programas que manipulan archivos de disco externos se complica con el hecho de que están implicados dos nombres diferentes: el *nombre del archivo externo*, que aparece en el directorio del disco en el que reside el archivo (nombre por el que se conoce el sistema operativo al archivo), y el *nombre del objeto flujo*, que es el *nombre interno* por el que su programa accede al archivo. En C++, la conexión entre estos dos nombres se realiza mediante el uso de una función especial, denominada `open`.

Para abrir y cerrar archivos se crean objetos `ifstream` y `ofstream`. Para comenzar a escribir en un archivo, se debe crear primero un objeto `ofstream` y, a continuación, asociar ese objeto con un archivo específico en su disco. Para utilizar objetos `ofstream`, debe asegurarse incluir el archivo `fstream` en su programa.

Nota: `fstream` *incluye* `iostream`*, por ello no es necesario incluir explícitamente* `iostream`.

Los archivos necesitan ser abiertos y cerrados. Estas operaciones se realizan con las funciones `open()` y `close()`. Las funciones implicadas en las operaciones de lectura y escritura son: `get/put` y `read/write` así como `seekp` y `seekg`.

EJERCICIOS

16.1. ¿Cuáles son los operadores de inserción y extracción? ¿Cuál es su misión?

16.2. Explicar las diferencias de los formatos de `cin.get()` y `cin.getline()`.

16.3. Escribir un programa que escriba en los cuatro objetos `iostream` estándar: `cin`, `cout`, `cerr` y `clog`.

16.4. Escribir un programa que solicite al usuario introducir su nombre completo y, a continuación, visualice en pantalla.

16.5. Escribir un programa que utilice las funciones `setf()`, `fill()` y `width()` que produzca la siguiente salida formateada:

```
Capítulo 5   Clases .................................. 300
Capítulo 6   Herencia ................................ 325
Capítulo 7   Flujos .................................. 355
```

16.6. Escribir el código para cada uno de los siguientes casos:

a) Imprimir en entero 12345 en un campo de 12 dígitos justificados a izquierda.

b) Imprimir 3.14159 en un campo de 12 dígitos con ceros precedentes.
c) Leer un entero en decimal e imprimirlo en octal.
d) Leer un entero en hexadecimal e imprimirlo en decimal.

16.7. Escribir una función `set_width(int w)` que establezca el campo con `cout` a w `columnas`.

16.8. Escribir un código C++ que lea una línea de texto y haga eco de la línea con todas las letras mayúsculas suprimidas.

16.9. Abrir un archivo de texto, leerlo línea a línea en un array de caracteres y escribirlo de nuevo en otro archivo.

16.10. Copiar un archivo, en modo binario, carácter a carácter.

16.11. Escribir un programa que lea un número arbitrario de enteros de un archivo que tiene el siguiente formato:

8
7
6
5
4
3
2
1

16.12. Escribir un programa que lea un texto del terminal y almacene el texto en un nuevo archivo de texto con el nombre `miarch.txt.` El nuevo archivo debe tener la misma estructura de línea que el archivo de texto escrito desde el terminal. Además, todas las letras minúsculas deben ser traducidas a letras mayúsculas.

16.13. Se dispone de un archivo `telefono.txt`, con nombres y números de teléfonos ordenados en orden alfabético. Escribir un programa que añada una nueva persona. El programa debe leer un nombre y número de teléfono de la nueva persona desde el terminal y, a continuación, insertar esta información en el lugar correcto del archivo de modo que permanezca ordenado. Sugerencia: utilice un archivo temporal.

16.14. La información acerca de un conjunto de personas se ha almacenado en un archivo binario para propósitos estadísticos. La información almacenada de cada persona es el nombre, altura, tamaño del zapato, edad y estado civil. Con el objeto de procesar los datos, el sexo de cada persona tiene que conocerse, aunque esta información no se incluye en el archivo. Escribir un programa que lea el archivo y crea dos archivos nuevos, uno que contiene sólo mujeres y otro con sólo hombres. Por cada persona del archivo, el programa debe pedir al operador si una persona es un hombre o una mujer.

16.15. Escribir un programa que permita crear un archivo inventario de los libros de una librería, así como calcular e imprimir el valor total del inventario. Los campos de cada libro deben ser, como mínimo, título, autor, ISBN, precio, cantidad, editorial.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 320).
2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

16.1. Escribir las sentencias necesarias para abrir un archivo de caracteres cuyo nombre y acceso se introduce por teclado en modo lectura. En el caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.

16.2. Un archivo de texto contiene enteros positivos y negativos. Utiliza operador de extracción >> para leer el archivo, visualizarlo y determinar el número de enteros negativos y positivos que tiene.

16.3. Escribir un programa que lea un texto del teclado lo almacene en un archivo binario y posteriormente vuelva a leer el archivo binario y visualice su contenido.

16.4. Se tiene un archivo de caracteres de nombre `"SALAS.DAT"`. Escribir un programa para crear el archivo `"SALAS.BIN"` con el contenido del primer archivo pero en modo binario.

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 321).
2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

16.1. Éstas son las operaciones básicas para realizar la apertura de un archivo que puede ser de entrada o de salida, luego debe ser un objeto de la clase `fstream`. Se trata de declarar una cadena de caracteres para leer el nombre del archivo, y abrir el archivo en modo entrada. En caso de que no pueda abrirse se debe crear de escritura. Una observación importante es que siempre se ha de comprobar si la apertura del archivo ha sido realizada con éxito, puesto que es una operación que realiza el sistema operativo para el programa queda fuera de nuestro control esta operación se realiza comprobando el valor de good.

La codificación de este ejercicio se encuentra en la página Web del libro.

16.2. Una vez que se conoce el funcionamiento del operador de extracción >> para la clase `cout`, es muy sencillo su uso para cualquier objeto de la clase `fstream`, ya que su funcionamiento es el mismo. El archivo abierto contiene números enteros, pero al ser un archivo de texto esos números están almacenados no de forma binaria sino como una cadena de caracteres que representan los dígitos y el signo del número en forma de secuencia de sus códigos ASCII binarios. Esto no quiere decir que haya que leer línea a línea y en cada una de ellas convertir las secuencias de códigos de caracteres a los números enteros en binario correspondiente, para almacenarlos así en la memoria. Este trabajo es el que realiza el operador de extracción >> cuando tiene una variable de tipo entero en su parte derecha. Es la misma operación de conversión que realiza el operador de extracción cuando lee secuencias de códigos de teclas desde la entrada estándar. El programa utiliza dos contadores para contar los positivos y negativos y una variable para leer el dato del archivo. El programa visualiza además el archivo.

La codificación de este ejercicio se encuentra en la página Web del libro.

CAPÍTULO 17

Listas enlazadas

Contenido

INTRODUCCIÓN

En este capítulo se comienza el estudio de las estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* —listas, vectores y tablas— y *estructuras*) en las que su tamaño en memoria se establece durante la compilación y permanece inalterable durante la ejecución del programa, las estructuras de datos dinámicas crecen y se contraen a medida que se ejecuta el programa.

La estructura de datos que se estudiará en este capítulo es la ***lista enlazada*** (*ligada* o *encadenada, «linked list»*) que es una colección de elementos (denominados *nodos*) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un «*enlace*» o «*puntero*». Las listas enlazadas son estructuras muy flexibles y con numerosas aplicaciones en el mundo de la programación.

CONCEPTOS CLAVE

- Eliminación de nodos en una lista enlazada.
- Estructura de una lista enlazada.
- Fundamentos teóricos de listas enlazadas.
- La clase *Pila* implementada mediante listas enlazadas.
- Lista doblemente enlazada.
- Operaciones en listas enlazadas.
- Recorrido de una lista.

17.1. FUNDAMENTOS TEÓRICOS

En capítulos anteriores, se han estudiado estructuras lineales de elementos homogéneos (listas, tablas, vectores) y se utilizaban *arrays* para implementar tales estructuras. Esta técnica obliga a fijar por adelantado el espacio a ocupar en memoria, de modo que cuando se desea añadir un nuevo elemento que rebase el tamaño prefijado del array, no es posible realizar la operación sin que se produzca un error en tiempo de ejecución. Ello se debe a que los arrays hacen un uso ineficiente de la memoria. Gracias a la asignación dinámica de variables, se pueden implementar listas de modo que la memoria física utilizada se corresponda con el número de elementos de la tabla. Para ello, se recurre a los **punteros** (*apuntadores*) que hacen un uso más eficiente de la memoria, como ya se ha visto con anterioridad.

Una **lista enlazada** es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente por un «enlace» o «puntero». La idea básica consiste en construir una lista cuyos elementos llamados **nodos** se componen de dos partes o *campos*: la primera parte o campo contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado *Dato*, *TipoElemento*, etc.) y la segunda parte o campo es un puntero que apunta al siguiente elemento de la lista.

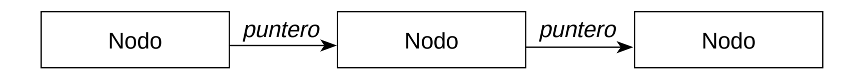


Figura 17.1. Lista enlazada (representación simple).

La representación gráfica más extendida es aquella que utiliza una caja (un rectángulo) con dos secciones en su interior. En la primera sección se escribe el elemento o valor del dato y en la segunda sección, el enlace o puntero mediante una flecha que sale de la caja y apunta al nodo siguiente.

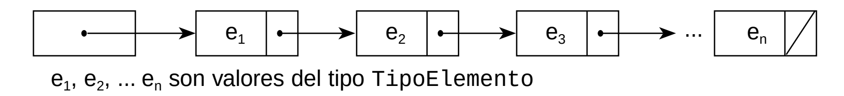


Figura 17.2. Lista enlazada (representación gráfica simple).

> Una **lista enlazada** consta de un número indeterminado de elementos y cada elemento tiene dos componentes (*campos*), un puntero al siguiente elemento de la lista y un valor, que puede ser de cualquier tipo.

Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos. Los enlaces también sitúan los nodos en una secuencia. En la Figura 17.2 los nodos forman una secuencia desde el primer elemento (e_1) al último elemento (e_n). El primer nodo se enlaza al segundo, el segundo se enlaza al tercero, y así sucesivamente, hasta llegar al último nodo. El último ha de ser representado de forma diferente para significar que este nodo no se enlaza a ningún otro. La Figura 17.3 muestra las diferentes representaciones gráficas que se utilizan para dibujar el nodo último.

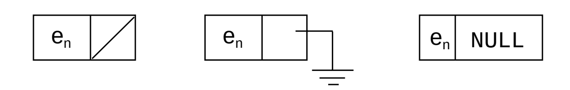


Figura 17.3. Diferentes representaciones gráficas del nodo último.

17.1.1. Clasificación de las listas enlazadas

Las listas se pueden dividir en cuatro categorías:

- *Listas simplemente enlazadas.* Cada nodo (elemento) contiene un único enlace que conecta ése al nodo siguiente o sucesor. La lista es eficiente en recorridos directos («*adelante*»).
- *Listas doblemente enlazadas.* Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su sucesor. La lista es eficiente tanto en recorrido directo («adelante») como en recorrido inverso («atrás»).
- *Lista circular simplemente enlazada.* Una lista enlazada simplemente en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular («en anillo»).
- *Lista circular doblemente enlazada.* Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (en anillo) tanto en dirección directa («adelante») como inversa («atrás»).

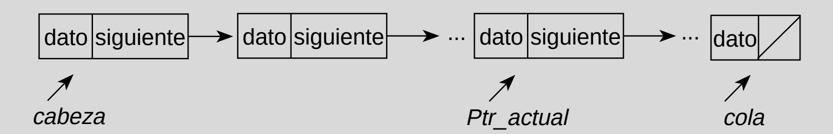
Por cada uno de estos cuatro tipos de estructuras de listas, se puede elegir una implementación basada en arrays o una implementación basada en punteros. Como ya se ha comentado, estas implementaciones difieren en el modo en que asigna la memoria para los datos de los elementos, cómo se enlazan juntos los elementos y cómo se accede a dichos elementos. De forma más específica, las implementaciones pueden hacerse con cualquiera de estas-asignaciones:

- Fija de memoria mediante array.
- Dinámica de memoria mediante punteros.

Dado que la asignación fija de memoria mediante arrays es más ineficiente, utilizaremos en este capítulo y siguientes la asignación de memoria mediante punteros, dejando como ejercicio al lector la implementación mediante arrays.

Conceptos importantes

Una lista enlazada consta de un conjunto de nodos. Un nodo consta de un campo dato y un puntero que apuntal al «siguiente» elemento de la lista.



El primer nodo, **frente**, es el nodo apuntado por **cabeza**. La lista encadena nodos juntos desde el frente al final (**cola**) de la lista. El final se identifica como el nodo cuyo campo puntero tiene el valor `NULL = 0`. La lista se recorre desde el primero al último nodo; en cualquier punto del recorrido la posición actual se referencia por el puntero `Ptr_actual`. En el caso en que la lista no contiene un nodo, el puntero cabeza es nulo.

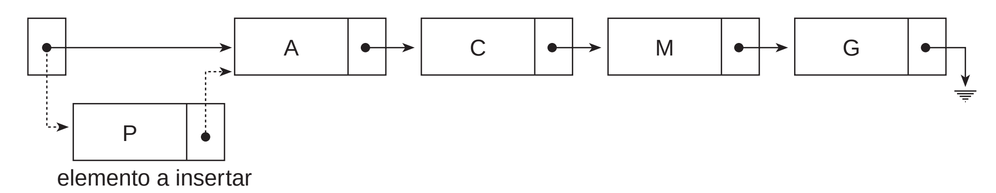
cabeza

`NULL`

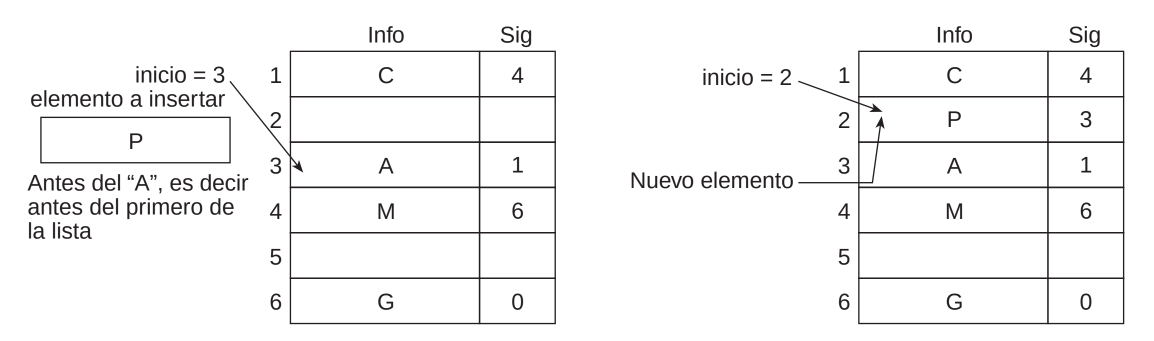
Ejemplo 17.1

Lista enlazada implementada con punteros y con arrays a la que se le añade un primer elemento.

Implementación con punteros:



Implementación con arrays:



Como se puede observar, tanto cuando se implementa con punteros como cuando se hace a través de arrays, la inserción de un nuevo elemento no requiere el desplazamiento de los que le siguen. Para observar la analogía entre ambas implementaciones se puede recurrir a representar la implementación con arrays de la siguiente forma:



17.2. OPERACIONES EN LISTAS ENLAZADAS

Una lista enlazada requiere unos controles para la gestión de los elementos contenidos en ellas. Estos controles se manifiestan en forma de operaciones que tendrán las siguientes tareas:

- *Inicialización o creación,* con declaración de los nodos.
- *Insertar elementos en una lista.*
- *Eliminar elementos de una lista.*
- *Buscar elementos de una lista* (comprobar la existencia de elementos en una lista).
- *Recorrer una lista enlazada.*
- *Comprobar si la lista está vacía.*

17.2.1. Declaración de un nodo

Una lista enlazada se compone de una serie de nodos enlazados mediante punteros. Cada nodo es una combinación de dos partes: un tipo de dato *(entero, real, doble, carácter,* etc.) y un enlace (*puntero*) al

siguiente nodo. En C++ se puede definir un nodo mediante un nuevo tipo de dato con las palabras reservadas `struct` o `class` que contiene las dos partes citadas.

```
struct Nodo              class Nodo {
{                        public :
   int dato;               int dato;
   Nodo * enlace;          Nodo *enlace;
};                         // constructor
                         };
```

La definición puede utilizar `struct`, que es un tipo especial de clase. De hecho, si se cambia la palabra reservada `struct` por `class`, la definición sigue siendo la misma. La diferencia sutil, como recordará el lector, es que en una *estructura* `struct` los miembros son siempre públicos mientras que en una *estructura* `class` los miembros son, por defecto, privados a menos que se ponga delante de ellos la palabra `public`.

Una estructura **`(struct)`** es un tipo especial de clase en la que sus miembros son todos públicos (a menos que se indique expresamente lo contrario). Los programadores de C++ suelen utilizar `struct` sólo cuando los miembros son públicos.

Una clase **`(class)`** es un tipo de dato definido por el usuario en la que todos sus miembros son privados a menos que se indique expresamente con la palabra reservada `public`, en cuyo caso, los miembros serán públicos.

Dado que los tipos de datos que se pueden incluir en una lista pueden ser de cualquier tipo (enteros, dobles, caracteres o incluso cadenas), con el objeto de que el tipo de dato de cada nodo se pueda cambiar con facilidad, se suele utilizar una sentencia `typedef` para definir el nombre de `elemento` como un sinónimo del tipo de dato de cada cadena. El tipo `elemento` se utiliza entonces dentro de la estructura nodo, como se muestra a continuación:

```
struct nodo
{
   typedef double elemento;
   elemento dato;
   nodo *enlace;
};
```

Entonces, si se necesita cambiar el tipo de elemento en los nodos, sólo tendrá que cambiar la sentencia de definición de tipos que afecta a `elemento`. Siempre que un programa necesite referirse al tipo de elemento, se puede utilizar la expresión `nodo::elemento`.

Ejemplo 17.2

Declaración de un nodo en C++.

En C++, se puede declarar un nuevo tipo de dato por un nodo mediante la palabra reservada `class` de la siguiente forma:

```
class Nodo
{
 public:
  int info;
  Nodo* sig:
};
```

```
typedef class Nodo
 {
  public:
    int info;
     Nodo *sig;
  }NODO;
```

```
typedef double Elemento
class Nodo
 {
   public:
    Elemento info;
    Nodo *sig;
 };
```

Ejemplo 17.3

Declaración e implementación de la clase `Nodo` que contiene una información de tipo elemento y el siguiente de tipo puntero a `Nodo`.

La clase `Nodo` tiene dos atributos protegidos que son el elemento `e`, y `Sig` que es un puntero a la propia clase `Nodo`. Ambos atributos sirven para almacenar la información del nodo, y la dirección del siguiente nodo. Se declaran como funciones miembro tres constructores. Estos constructores son: el constructor por defecto; el constructor que inicializa el atributo `x` del `Nodo` al valor de `x`, y pone el atributo `Sig` a `NULL`; el constructor, que inicializa los dos atributos del `Nodo`. Las funciones miembro encargadas de `Obtener` y `Poner` tanto el elemento `e` como el puntero `Sig` son: `Telemento OE(); void PE(Telemento e); Nodo * OSig();` y `void PSig(Nodo *p);`. Por último, la función miembro destructor, se declara por defecto. El tipo elemento de la clase `Nodo` es un entero, pero al estar definido en un `typedef`, puede cambiarse de una forma sencilla y rápida para que almacene cualquier otro tipo de información.

```
typedef int Telemento;

class Nodo
{
    protected:
        Telemento e;
        Nodo *Sig;
    public:
        Nodo(){}                                        // constructor vacío
        Nodo (Telemento x);                                   // constructor
        Nodo(Telemento x, Nodo* s);                           // Constructor
        ~Nodo(){}                                // destructor por defecto
        Telemento OE();                                 // Obtener elemento
        void PE(Telemento e);                             // Poner elemento
        Nodo * OSig();                                 // Obtener siguiente
        void PSig( Nodo *p);                             // Poner siguiente
 };

Nodo::Nodo(Telemento x)
{                         // constructor que inicializa e a x y Sig a NULL
  e = x;
  Sig = NULL;
}

Nodo(Telemento x, Nodo* s)
{                            // constructor que inicializa e a x y Sig a s
  e = x;
  Sig = s;
}
```

```
Telemento Nodo::OE()
{                                     //  obtiene una copia del atributo e
  return e;
}

void Nodo::PE(Telemento x)
{                                                 // pone el atributo e a x
  e = x;
}

Nodo* Nodo::OSig()
{                                   //  obtiene una copia del atributo Sig
  return Sig;
}

void Nodo:: PSig(Nodo *p)
{                                               // Pone el atributo Sig a p
  Sig = p;
}
```

17.2.2. Puntero de cabecera y cola

Normalmente, los programas no declaran realmente variables de nodos. En su lugar, cuando se construye y manipula una lista enlazada, a la lista se accede a través de uno o más *punteros* a los nodos. El acceso más frecuente a una lista enlazada es a través del primer nodo de la lista que se llama **cabeza** o **cabecera** de la lista. Un puntero al primer nodo se llama **puntero cabeza**. En ocasiones, se mantiene también un puntero al último nodo de una lista enlazada. El último nodo es la **cola** de la lista, y un puntero al último nodo es el **puntero cola**. También se pueden mantener punteros a otros nodos de una lista enlazada.

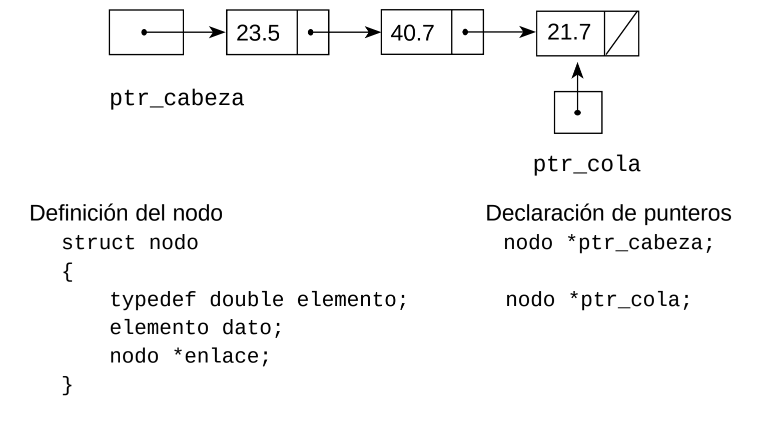


Figura 17.4. Declaraciones de tipos en lista enlazada.

Cada puntero a un nodo debe ser declarado como una variable puntero. Por ejemplo, si se mantiene una lista enlazada con un puntero de cabecera y otro de cola, se deben declarar dos variables puntero:

```
nodo *ptr_cabeza;
nodo *ptr_cola;
```

La **construcción y manipulación de una lista enlazada** requiere el acceso a los nodos de la lista a través de uno o más punteros a nodos. Normalmente, un programa incluye un puntero al primer nodo (*cabeza*) y un puntero al último nodo (*cola*).

De cualquier forma, el último elemento de la lista contiene un valor de 0, esto es, un puntero nulo (`NULL`) que señala el final de la lista.

17.2.3. El puntero nulo

La Figura 17.5 muestra una lista con un puntero cabeza y un puntero nulo al final sobre el que se ha escrito la palabra `NULL`. La palabra `NULL` representa el **puntero nulo**, que es una constante especial de C++. Se puede utilizar el puntero nulo para cualquier valor de puntero que no apunte a ningún sitio. El puntero nulo se utiliza, normalmente, en dos situaciones:

- Usar el puntero nulo en el campo enlace o siguiente del nodo final de una lista enlazada.
- Cuando una lista enlazada no tiene ningún nodo, se utiliza el puntero `NULL` como puntero de cabeza y de cola. Esta lista se denomina **lista vacía**.

En un programa, el puntero nulo se puede escribir como `NULL`, que es una constante de la biblioteca estándar `stdlib.h`[1]. El puntero nulo se puede asignar a una variable puntero con una sentencia de asignación ordinaria. Por ejemplo,

```
nodo *ptr_cabeza;
ptr_cabeza = NULL ;      // incluir archivo stdlib.h
```

• →	4.15 →	5.25 →	71.5 →	10.5 NULL

ptr_cabeza

Figura 17.5. Puntero `NULL`.

El puntero de cabeza y de cola en una lista enlazada puede ser `NULL`, lo que indicará que la lista es vacía (no tiene nodos). Éste suele ser un método usual para construir una lista. Cualquier función que se escribe para manipular listas enlazadas debe poder manejar un puntero de cabeza y un puntero de cola nulos.

17.2.4. El operador -> de selección de un miembro

Si `p` es un puntero a una clase (o una estructura) y `m` es un miembro de la clase, entonces `p -> m` accede al miembro `m` de la estructura apuntado por `P`.

[1] A veces algunos programadores escriben el puntero nulo como 0, pero pensamos que es un estilo más claro escribirlo como `NULL`.

El símbolo «`->`» se considera como un operador simple (en vez de compuesto, al constar de dos símbolos independientes: «`-`» y «`>`»). Se denomina *operador de selección de miembro* o también *operador de selección de componente*. De modo visual el operador `P -> m` recuerda a una flecha que apunta del puntero `p` al objeto que contiene al miembro `m`.

Suponiendo que un programa ha de construir una lista enlazada y crear un puntero de cabecera `ptr_cabeza` a un nodo `Nodo`, el operador `*` de indirección aplicado a una variable puntero representa el contenido del nodo apuntado por `ptr_cabeza`. Es decir, `*ptr_cabeza` es un tipo de dato `Nodo`.

Al igual que con cualquier objeto, se puede acceder a los dos miembros de `*ptr_cabeza` en la Figura 17.5. Por ejemplo, la sentencia siguiente escribe los datos del nodo cabecera.

```
cout << (*ptr_cabeza).dato
(*ptr_cabeza)   miembro dato del nodo apuntado por  ptr_cabeza
```

Precaución

Los paréntesis son necesarios alrededor de la primera parte de la expresión `(*ptr_cabeza)` ya que los operadores unitarios que aparecen a la derecha tienen prioridad más alta que los operadores unitarios que aparecen en el lado izquierdo (el asterisco de indirección).

Sin los paréntesis, el significado de `cabeza_ptr` producirá un error de sintaxis, al intentar evaluar `ptr_cabeza.dato` antes de la *indirección* o *desreferencia*.

`p -> m` *significa lo mismo que* `(*p).m`

Utilizando el operador de selección se pueden imprimir los datos del primer nodo de la lista

```
cout << ptr_cabeza -> dato;
```

Error

Uno de los errores típicos en el tratamiento de punteros es escribir la expresión `*p` o bien `p->` cuando el valor del puntero p es el puntero nulo, ya que como se sabe el puntero nulo no apunta a nada.

17.2.5. Construcción de una lista

Un algoritmo para la creación de una lista enlazada entraña los siguientes pasos:

Paso 1. Declarar el tipo de dato y el puntero de cabeza o primero.
Paso 2. Asignar memoria para un elemento del tipo definido anteriormente utilizando el operador `new`; la dirección del nuevo elemento es `ptr_nuevo`.
Paso 3. Crear iterativamente el primer elemento (cabeza) y los elementos sucesivos de una lista enlazada simplemente.
Paso 4. Repetir hasta que no haya más entrada para el elemento.

Ejemplo 17.4

Crear una lista enlazada de elementos que almacenen datos de tipo entero.

Un elemento de la lista se puede definir con la ayuda de la clase siguiente:

```
class Elemento {
public :
   Elemento * siguiente;
   int dato;
   Elemento (Elemento *n, int d): siguiente(n), datos(d) {}
};
```

En la clase `Elemento` hay dos miembros dato: `siguiente,` que es un puntero al siguiente elemento, y `dato,` que contiene el valor del elemento de la lista. También se define un constructor que se llama cuando se crean nuevos elementos de la lista que se añaden a la clase `Elemento`. Por simplicidad, se han situado todos los miembros en la sección `public` (pública) en donde serán visibles.

El siguiente paso para construir la lista es declarar la variable `Primero` que apuntará al primer elemento de la lista:

```
Elemento *Primero = NULL         // o bien = 0
```

El puntero `Primero` (también se puede llamar `Cabeza`) se ha inicializado a un valor nulo, lo que implica que la lista está vacía (no tiene elementos). Ahora se crea un elemento de la lista y se sitúa en la lista con la operación:

```
Primero = new Elemento (Primero,5);
```

5 NULL

Primero

El puntero `Primero` apunta al nuevo elemento, que se inicializa por el constructor de la clase `Elemento`. Dado que el puntero `Primero` tenía un valor de nulo (cero) el campo `siguiente` del nuevo elemento tomará el valor nulo.

Si ahora se desea añadir un nuevo elemento con un valor de 3 y situarlo en el primer lugar de la lista se escribe simplemente:

```
Primero = new Elemento (Primero,3);
```

3 5 NULL

Por último, para obtener una lista `4`, `3`, `5` se habría de ejecutar

```
Primero = new Elemento (Primero,4);
```

4 3 5

Ejemplo 17.5

Clase lista simple con constructor, destructor, y funciones miembro para poner y obtener el puntero al primer nodo de la lista.

```
#include <cstdlib>
#include <iostream>
using namespace std;

class ListaS                                //clase lista simplemente enlazada
{
  protected:
      Nodo *p;
  public:
      ListaS();                                           // constructor
      ~ListaS();                                           // destructor
      Nodo * Op();                                  // Obtener el puntero
      void Pp( Nodo *p1);                             // Poner el puntero
                                              // otras funciones miembro
                                                              //......
};

ListaS::ListaS()
{
  p = NULL;
}

ListaS::~ListaS(){};
Nodo * ListaS:: Op()
{
    return p;
}

void ListaS:: Pp( Nodo *p1)
{
   p = p1;
}

int main(int argc, char *argv[])
{
    ListaS l1;
    ......
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

17.2.6. Insertar un elemento en una lista

El algoritmo empleado para añadir o insertar un elemento en una lista enlazada varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento último).

- Antes de un elemento especificado.
- Después de un elemento especificado.

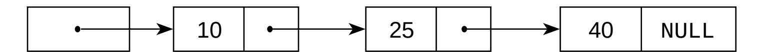
Insertar un nuevo elemento en la cabeza de una lista

Aunque normalmente se insertan nuevos datos al final de una estructura de datos, es más fácil y más eficiente insertar un elemento nuevo en la cabeza de una lista. El proceso de inserción se puede resumir en este algoritmo:

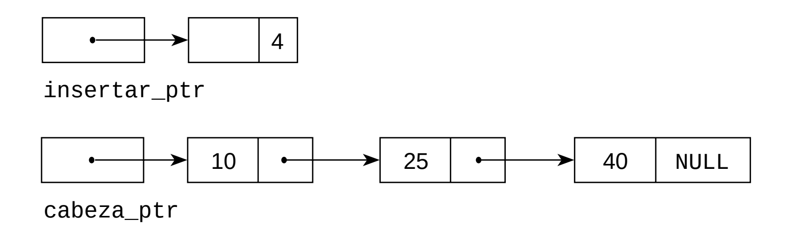
- Asignar un nuevo nodo apuntado por `insertar_ptr` que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista.
- Situar el nuevo elemento en el campo `dato` (`Info`) del nuevo nodo.
- Hacer que el campo enlace `siguiente` del nuevo nodo apunte a la cabeza (primer nodo) de la lista original.
- Hacer que `cabeza_ptr` (puntero cabeza) apunte al nuevo nodo que se ha creado.

Ejemplo 17.6

Una lista enlazada contiene tres elementos, 10, 25 y 40. Insertar un nuevo elemento 4 en cabeza de la lista.



Pasos 1 y 2



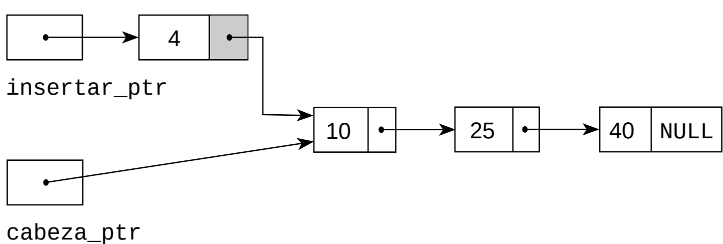
Código C++

```
insertar_ptr = new Nodo;    // se asigna un nuevo nodo
insertar_ptr -> dato = entrada
```

Paso 3

El campo enlace (`siguiente`) del nuevo nodo apunta a la cabeza actual de la lista

Código C++ `insertar_ptr -> siguiente = cabeza_ptr;`



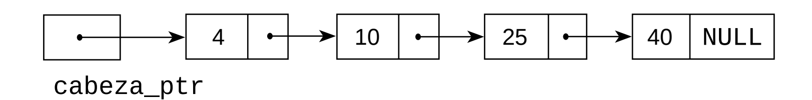
Paso 4

Se cambia el puntero de cabeza para apuntar al nuevo nodo creado; es decir, el puntero de cabeza apunta al mismo sitio que apunta `insertar_ptr`.

Código C++ `cabeza_ptr = insertar_ptr;`

insertar_ptr
4
10
25
40 NULL
cabeza_ptr

En este momento, la función de insertar un elemento en la lista termina su ejecución y la variable local `insertar_ptr` desaparece y sólo permanece el puntero de cabeza `cabeza_ptr` que apunta a la nueva lista enlazada.



El código fuente de la función `InsertarCabezaLista` es

```
void InsertarCabezaLista(Nodo& cabeza_ptr,
                        const Nodo::Item& entrada)
{
   Nodo *insertar_ptr;
   insertar_ptr = new Nodo;          // asigna nuevo nodo
   insertar_ptr -> dato = entrada;   // pone elemento en nuevo
                                     // nodo
   insertar_ptr -> siguiente =
                   cabeza_ptr ;      // enlaza nuevo nodo
                                     // al frente de la lista
   cabeza_ptr = insertar_ptr ;       // mueve puntero cabeza
                                     // y apunta al nuevo nodo
}
```

Caso particular

La función `InsertarCabezaLista` actúa también correctamente si se trata el caso de añadir un primer nodo o elemento a una lista vacía. En este caso, y como ya se ha comentado `cabeza_ptr` apunta a `NULL` y termina apuntando al nuevo nodo de la lista enlazada.

Ejemplo 17.7

Función miembro de la clase `ListaS` que añade un nuevo elemento `e` a la lista.

Se supone las declaraciones de los Ejemplos 17.3 y 17.4 de las clases `Nodo` y `ListaS`. En la zona pública de la clase `ListaS` se debe incluir el prototipo de la función miembro.

```
void AnadePL(Telemento e);        // prototipo dentro de la clase ListaS

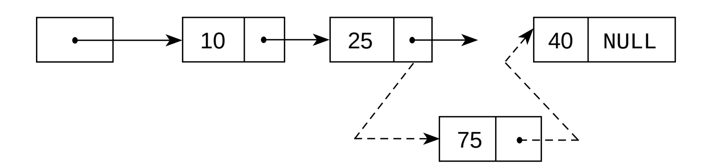
void ListaS::AnadePL(Telemento e)                              // código
{
  Nodo *aux;
  aux = new Nodo(e);                   // reserva de memoria, y asigna e
  aux -> PSig(p);                      // poner p en el siguiente de aux
  p =  aux;                                    //Pp(aux); poner p  a aux
}
```

Inserción de un nuevo nodo que no está en la cabeza de lista

La inserción de un nuevo nodo no siempre se realiza al principio (en cabeza) de la lista. Se puede insertar en el centro o al final de la lista.

Ejemplo 17.8

Se desea insertar un nuevo elemento 75 entre el elemento 25 y el elemento 40 en la lista enlazada 10, 25, 40.

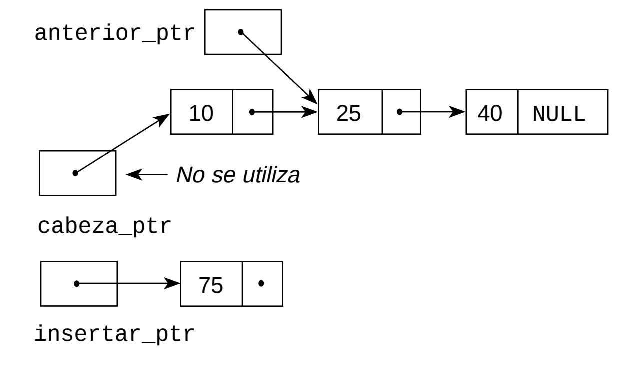


El algoritmo de la nueva operación insertar requiere las siguientes etapas:

- Asignar el nuevo nodo apuntado por el puntero `insertar_ptr`.
- Situar el nuevo elemento en el campo *dato* del nuevo nodo.
- Hacer que el campo enlace `siguiente` del nuevo nodo apunte al nodo que va después de la posición del nuevo nodo (o bien a `NULL` si no hay ningún nodo después de la nueva posición).
- Crear un puntero al nodo que está antes de la posición deseada para el nuevo nodo y hacer que `anterior_ptr` -> `siguiente` apunte al nuevo nodo que se acaba de crear.

Etapas 1 y 2

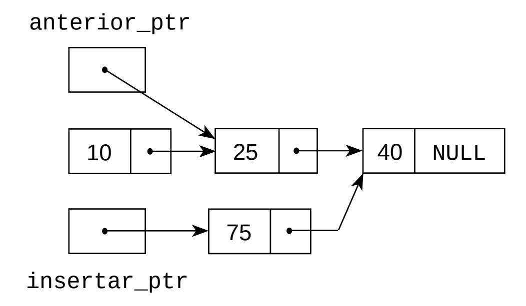
Se crea un nuevo nodo que contiene a 75



Código C++

```
insertar_ptr = new Nodo;
insertar_ptr -> dato = entrada;
```

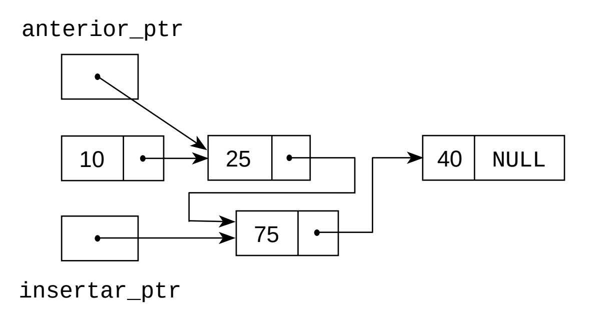
Etapa 3



Código C++

```
insertar_ptr -> siguiente = anterior_ptr ->
                                      siguiente
```

Etapa 4



Después de ejecutar todas las sentencias de las sucesivas etapas la nueva lista comenzaría en el nodo 10, seguiría en 25, 75 y, por último, 40.

Programa C++

```
void InsertarLista(Nodo* anterior_ptr,const Nodo::Item& entrada)
{
   Nodo *insertar_ptr;

   insertar_ptr = new Nodo;
   insertar_ptr -> dato = entrada;
   insertar_ptr -> siguiente = anterior_ptr -> siguiente;
   anterior_ptr -> siguiente = insertar_ptr;
}
```

Inserción al final de la lista

La inserción al final de la lista es menos eficiente debido a que, normalmente, no se tiene un puntero al último elemento de la lista y entonces se ha de seguir la traza desde la cabeza de la lista hasta el último nodo de la lista y, a continuación, realizar la inserción. Cuando `ultimo` es una variable puntero que apunta al último nodo de la lista, las sentencias siguientes insertan un nodo al final de la lista.

```
ultimo -> siguiente = new nodo;
ultimo -> siguiente -> dato = entrada;
ultimo -> siguiente -> siguiente = NULL;
```

La primera sentencia anterior asigna un nuevo nodo que está apuntado por el campo `siguiente` al último nodo de la lista (antes de la inserción) de modo que el nuevo nodo ahora es el último nodo de la lista. La tercera sentencia establece el campo `siguiente` del nuevo último nodo a `NULL`.

Ejemplo 17.9

Segmento de código que añade un nuevo elemento `e` a la lista, entre las posiciones `anterior_ptr` y `pos`. (Si `pos` es `NULL`, es el último de la lista.)

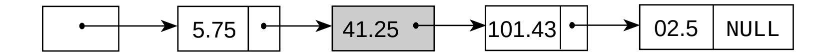
Se supone declarada la clase `Nodo`, y que previamente, se ha colocado el puntero a `Nodo aux` apuntado al nodo anterior donde hay que colocarlo (se supone que al no ser el primero, siempre existe), y el puntero a `Nodo pos` apuntando al nodo que va después.

```
Nodo aux, *pos, *insertar_ptr;

// búsqueda omitida.
insertar_ptr = new Nodo(x);
insertar_ptr->PSig(pos);
anterior_ptr->PSig(aux);
```

17.2.7. Búsqueda de un elemento

Dado que una función en C++ puede devolver un puntero, el algoritmo que sirva para localizar un elemento en una lista enlazada puede devolver un puntero a ese elemento.



La función `BuscarLista` utiliza una variable puntero denominada `Indice` que va recorriendo la lista nodo a nodo. Mediante un bucle, `Indice` apunta a los nodos de la lista de modo que si se encuentra el nodo buscado, se devuelve un puntero al nodo buscado con la sentencia de retorno (`return`); en el caso de no encontrarse el nodo buscado la función debe devolver `NULL` (`return NULL`).

Código C++

```
Nodo* BuscarLista (Nodo* cabeza_ptr, const Nodo::Item destino)
// cabeza_ptr, puntero de cabeza de una lista enlazada
// destino, parámetro que representa al nodo especificado
// Indice, valor de retorno: puntero que apunta al primer
// nodo que contiene el destino (elemento buscado)
// Si no existe el nodo, se devuelve puntero nulo
{
   Nodo *Indice;

   for (Indice = cabeza_ptr; Indice != NULL; Indice = Indice ->
        Siguiente)
```

```
        if (destino == Indice -> dato)
          return Indice;
   return NULL;
}
```

Ejemplo 17.10

Encontrar un nodo dada su posición en una lista enlazada.

El nodo o elemento se especifica por su posición en la lista; para ello se considera posición 1, la correspondiente al nodo de cabeza; posición 2, la correspondiente al siguiente nodo, y así sucesivamente.

El algoritmo de búsqueda del elemento comienza con el recorrido de la lista mediante un puntero `Indice` que comienza apuntando al nodo cabeza de la lista. Un bucle mueve el `Indice` hacia adelante el número correcto de sitios (lugares). A cada iteración del bucle se mueve el puntero `Indice` un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada e `Indice` apunta al nodo correcto. El bucle también se puede terminar si `Indice` apunta a `NULL`, lo que indicará que la posición solicitada era más grande que el número de nodos de la lista.

Código C++[2]

Para poder utilizar `assert` ha de incluir al principio del programa la directiva de inclusión `#include <assert.h>`.

```
Nodo* BuscarPosicion(Nodo *cabeza_ptr, size_t posicion)
// El programa que manipula esta función ha de incluir
// biblioteca stdlib.h (para implementar tipo size_t)
// biblioteca assert.h (para implementar función assert)
{
   Nodo *indice;
   size_t i;

   assert (0 < posicion)   // posicion ha de ser mayor que 0
   índice = cabeza_ptr;
   for (i = 1;(i < posición) && (indice!= NULL); i++)
       indice = indice -> siguiente;
   return indice;
}
```

Ejemplo 17.11

Búsqueda de un elemento en una lista enlazada implementada con clases.

La función miembro `Buscar` retorna un puntero al nodo que contiene la primera aparición del dato `x`, en la lista en el caso de que se encuentre, y `NULL` en otro caso. Se usa un puntero al nodo `pos` que se inicializa con el primer puntero de la lista `p`. Un bucle `while`, itera, mientras queden datos en la lista y no haya encontrado el elemento que se busca. Se incluye, además, la función miembro `Buscar1`, que realiza la misma búsqueda sin utilizar interruptor `enc` de la búsqueda y con un bucle `for`.

---

[2] La Biblioteca ANSI C++ proporciona una biblioteca `<assert.h>` que contiene una función principal `assert` que tiene un argumento (puede ser una expresión entera). La función `assert` evalúa la expresión, si el resultado es verdadero se realiza la acción prevista; si el resultado es falso se visualiza un mensaje de error y se detiene el programa.

Para poder utilizar `assert` ha de incluir al principio del programa la directiva de inclusión `#include <assert.h>`.

```
Nodo * ListaS::Buscar(Telemento x)
{
  Nodo *pos = p;
  bool enc = false;

   while (!enc && pos)
     if (pos->OE() != x)
       pos = pos->OSig();
     else enc = true;

    if (enc)         // Encontrado
       return pos;
    else
        return NULL;
}

Nodo* ListaS::Buscar1(Telemento x)
{
   Nodo *pos = p;
   for (; pos != NULL; pos = pos->OSig())
      if (x == pos-> OE())
          return pos;
  return NULL;
}
```

Ejemplo 17.12

Función miembro de la clase `ListaS` *que añade el elemento* `x` *en la posición* `posi` *que recibe como parámetro.*

Se realiza una búsqueda en la lista enlazada, quedándose, además de con la posición `pos`, con un puntero al nodo anterior `ant`. El bucle que realiza la búsqueda lleva un contador que va contando el número de elementos que se recorren, para que cuando se llegue al límite `pos`, poder terminar la búsqueda. A la hora de añadir el nodo a la lista, se observa las dos posibilidades existentes en la inserción en una lista enlazada: primero de la lista o no primero (incluye al último).

```
void ListaS::AnadeposL(Telemento x, int posi)
{
 Nodo *ant = NULL, *pos = p, *aux = new Nodo(x);
 int c = 0;                               // contador de nodos de la lista
 while (pos && c < posi)
{
    ant = pos;
    pos = pos->OSig();
    c++;
}
if (c  < posi)
   {
     aux->PSig(pos);
   if ( ant )                                // centro o final de la lista
       ant->PSig(aux);
```

```
        else p = aux;                                // primero de la lista
      }
    }
```

17.2.8. Eliminar elementos de una pila

El algoritmo para eliminar un nodo que contiene un dato se puede expresar en estos pasos:

- Buscar el nodo que contiene el dato y que esté apuntado por `pos`. Hay que tener la dirección del nodo a eliminar y la del inmediatamente anterior `ant`.
- El puntero `Sig` del nodo anterior `ant` ha de apuntar al `Sig` del nodo a eliminar.
- Si el nodo a eliminar es el primero de la lista se modifica el atributo `p` de la clase lista que apunta al primero para que tenga la dirección del nodo `Sig` del nodo a eliminar `pos`.
- Se libera la memoria ocupada por el nodo `pos`.

Ejemplo 17.13

Eliminación de un nodo de una lista implementada con un clase `ListaS`.

El código que se presenta, corresponde a la función miembro `Borrar` de la clase `ListaS` implementada en ejemplos anteriores.

```
void ListaS::Borrar(Telemento x)
{
    Nodo *ant = NULL, *pos =p;
      // búsqueda del nodo a borrar
    ......
      // se va a borrar el nodo apuntado por pos

    if(ant != NULL)
       ant->PSig(pos->OSig());                  // no es el primero
    else
        p = pos->OSig();                          // es el primero
    pos->PSig(NULL);
    delete pos;
 }
```

Ejemplo 17.14

Función miembro de la clase `ListaS` que elimina la primera aparición del elemento `x` de la lista si es que se encuentra.

Primero, se realiza la búsqueda de la primera aparición del elemento `x`, quedándose, además, con un puntero `ant` que apunta al nodo inmediatamente anterior, en caso de que exista. Si se ha encontrado el elemento a borrar en la posición `pos`, se diferencian las dos posibilidades que hay en el borrado en una lista enlazada simple. Ser el primer elemento, en cuyo caso el puntero anterior apunta a `NULL`, o no serlo. Si el elemento a borrar es el primero de la lista, hay que mover el primer puntero de la lista `p` al nodo siguiente de `p`. En otro caso, hay que poner el puntero siguiente del nodo anterior en el puntero siguiente de la posición donde se encuentre.

```
void ListaS::BorrarL(Telemento x)
 {
   Nodo *ant = NULL, *pos = p;
   bool enc = false;
   while (! enc && pos)                                   //búsqueda
   if (pos->OE()!= x)
   {
     ant = pos;
     pos = pos->OSig();
   }
   else enc =true;
   if (enc)                         // borrar la primera aparición
   {
       if (ant)                                         // no primero
        ant->PSig(pos->OSig());
      else                                    // borrado del primero
         p= p->OSig();
      pos->PSig(NULL);
      delete pos;
   }
}
```

17.3. LISTA DOBLEMENTE ENLAZADA

Hasta ahora el recorrido de una lista se realizaba en sentido directo (*adelante*) o, en algunos casos, en sentido inverso (*hacia atrás*). Sin embargo, existen numerosas aplicaciones en las que es conveniente poder acceder a los elementos o nodos de una lista en cualquier orden. En este caso se recomienda el uso de una **lista doblemente enlazada**. En tal lista, cada elemento contiene dos punteros, aparte del valor almacenado en el elemento. Un puntero apunta al siguiente elemento de la lista y el otro puntero apunta al elemento anterior. La Figura 17.6 muestra una lista doblemente enlazada y un nodo de dicha lista.

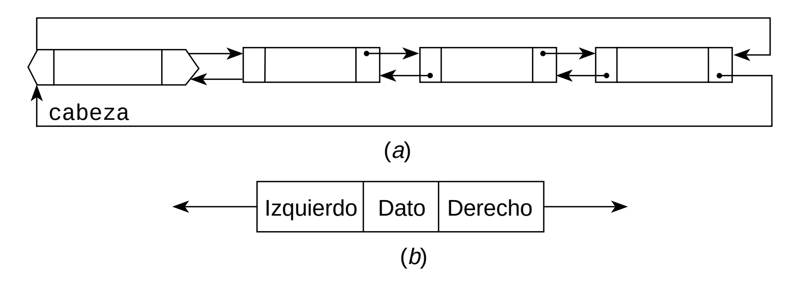


Figura 17.6. Lista doblemente enlazada. (*a*) Lista con tres nodos; (*b*) nodo.

Existe una operación de *insertar* y *eliminar* (borrar) en cada dirección. La Figura 17.7 muestra el problema de insertar un nodo *p* a la derecha del nodo actual. Deben asignarse cuatro nuevos enlaces.

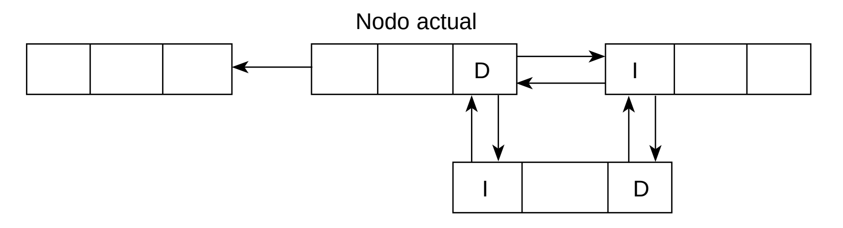


Figura 17.7. Inserción de un nodo en una lista doblemente enlazada.

En el caso de eliminar (borrar) un nodo de una lista doblemente enlazada es preciso cambiar dos punteros.

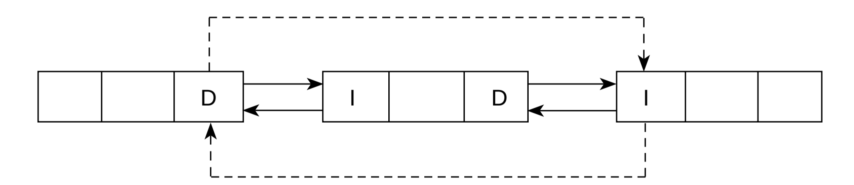


Figura 17.8. Eliminación de un nodo en una lista doblemente enlazada.

17.3.1. Declaración de una lista doblemente enlazada

Una lista doblemente enlazada con valores de tipo `int` necesita dos punteros, el valor del campo dato y en el caso de ser representada con `class` un constructor con el que se pueden construir los nuevos elementos. Este constructor funcionará como constructor por defecto.

```
class Elemento
{
   public :
   Elemento *adelante, *atras;
   int dato;                  // dato de tipo entero
   Elemento (Elemento *f = 0, Elemento *b = 0, int d = 0)
     : adelante(f), atras(b), dato(d) {}
};
```

Otra forma de representar una lista enlazada puede ser con una estructura (`struct`) del modo siguiente:

```
struct Nodo
{
   typedef int Elemento;
   Elemento dato;
   Nodo *adelante;
   Nodo *atras;
};
```

17.3.2. Inserción de un elemento en una lista doblemente enlazada

El algoritmo empleado para añadir o insertar un elemento en una lista doble varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser: en la cabeza (elemento primero) de la lista; en el final de la lista (elemento último); antes de un elemento especificado, o bien después de un elemento especificado.

Inserción de un nuevo elemento en la cabeza de una lista doble

El proceso de inserción se puede resumir en este algoritmo:

- Asignar memoria a un nuevo nodo apuntado por `nuevo` que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista doble y situar en el nodo `nuevo` el elemento `e` que se va a insertar.

- Hacer que el campo enlace `Sig` del nuevo nodo `nuevo` apunte a la cabeza (primer nodo `p`) de la lista original, y que el campo enlace `Ant` del nodo cabeza `p` apunte al nuevo nodo `nuevo` si es que existe. En caso de que no exista no hacer nada.
- Hacer que cabeza (puntero de la lista `p`) apunte al nuevo nodo que se ha creado.

Ejemplo 17.15

Función miembro de la clase `ListaD` *que añade un nuevo elemento a la lista doble como primer elemento.*

```
void ListaD::InsertaP(Telemento e)
{
  NodoD *nuevo = new NodoD(el);
  nuevo->Psig(p);
  if(p)                               // si existe p que apunte a un nodo
     p->Pant(nuevo);
  p = nuevo;
}
```

Inserción de un nuevo nodo que no está en la cabeza de lista

La inserción de un nuevo nodo en una lista doblemente enlazada se puede realizar en un nodo intermedio o al final de ella. El algoritmo de la nueva operación insertar *requiere* las siguientes etapas:

- Buscar la posición donde hay que insertar el dato, dejando los punteros a `NodoD: ant` apuntando al nodo inmediatamente anterior (siempre existe), y `pos` al que debe ser el siguiente en caso de que exista (`NULL` en caso de que no exista).
- Asignar memoria al nuevo nodo apuntado por el puntero `nuevo`, y situar el nuevo elemento `e` como atributo del nuevo nodo `nuevo`.
- Hacer que el atributo `Sig` del nuevo nodo `nuevo` apunte al nodo `pos` que va después de la posición del nuevo nodo `ptrnodo` (o bien a `NULL` en caso de que no haya ningún nodo después de la nueva posición). El atributo `Ant` del nodo siguiente `ptrnodo` al que ocupa la posición del nuevo nodo `nuevo` que es `pos`, tiene que apuntar a `nuevo` si es que existe. En caso de que no exista no hacer nada.
- Hacer que el atributo `Sig` del puntero `ant` apunte al nuevo nodo `nuevo`. El atributo `Ant` del nuevo nodo `nuevo` ponerlo apuntando a `ant`.

Ejemplo 17.16

Segmento de código de una función miembro de la clase `ListaD` *que añade un nuevo elemento a la lista doble en una posición que no es la primera.*

```
void ListaD::InsertaListadoble(Telemento e)
{
  NodoD *nuevo, *ant=NULL, *pos=p;
                       // búsqueda de la posición donde colocar el dato e
                       // se sabe que no es el primero de la lista doble

  aux = new NodoD(x);
  if (!pos)                                          //último y no primero
  {
```

```
      ant->PSig(nuevo);
      nuevo->PAnt(ant);
   }
   else                                          //no primero y no último
   {
   nuevo->PSig(pos);
   nuevo->PAnt(ant);
   ant->PSig(nuevo);
   pos->PAnt(nuevo);
   }
}
```

17.3.3. Eliminación de un elemento en una lista doblemente enlazada

El algoritmo para eliminar un nodo que contiene un dato es similar al algoritmo de borrado para una lista simple. Ahora la dirección del nodo anterior se encuentra en el puntero `ant` del nodo a borrar. Los pasos a seguir son:

- Búsqueda del nodo que contiene el dato. Se ha de tener la dirección del nodo a eliminar y la dirección del anterior (`ant`).
- El atributo `Sig` del nodo anterior (`ant`) tiene que apuntar al atributo `Sig` del nodo a eliminar, `pos` (esto en el caso de no ser el nodo primero de la lista). En caso de que sea el primero de la lista el atributo `p` de la lista debe apuntar al atributo `Sig` del nodo a eliminar `pos`.
- El atributo `Ant` del nodo siguiente a borrar tiene que apuntar al atributo `Ant` del nodo a eliminar, esto en el caso de no ser el nodo ultimo. En el caso de que el puntero a eliminar sea el último no hacer nada.
- Por último, se libera la memoria ocupada por el nodo a eliminar `pos`.

Ejemplo 17.17

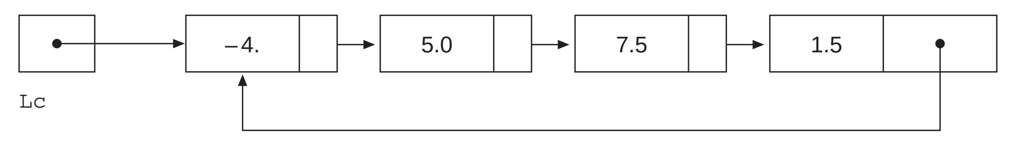
Segmento de código de una función miembro de la clase `ListaD` *que borra un elemento en una lista doblemente enlazada.*

```
void ListaD::Borrarelemento(Telemento x)
{
  NodoD *ant = NULL, *pos = p;
                                       // búsqueda del nodo a borrar omitida
                                                    // NodoD a borrar en pos
  if(!ant && !pos->OSig())                         // es el primero y último
  {
    p = NULL;                                //la lista doble se queda vacía
  }
  else if (!ant)                                      // primero y no último
  {
    pos->OSig()->PAnt(NULL);
    p = pos->OSig();
    pos->PSig(NULL);
  }
  else if (!pos->OSig())                               //ultimo y no primero
  {
```

```
      ant->PSig(NULL);
      pos->PAnt(NULL);
      }
      else                                            // no primero y no último
      {
      ant->PSig(pos->OSig());
      pos->OSig()->PAnt(ant);
      pos->PAnt(NULL);
      pos->PSig(NULL);
     }
     delete pos;
  }
```

17.4. LISTAS CIRCULARES

En las listas lineales simples o en las dobles siempre hay un primer nodo y un último nodo que tiene el atributo de enlace a nulo (`NULL`). Una lista circular, por propia naturaleza no tiene ni principio ni fin. Sin embargo, resulta útil establecer un nodo a partir del cual se acceda a la lista y así poder acceder a sus nodos insertar, borrar etc.



Lista circular

Inserción de un elemento en una lista circular

El algoritmo empleado para añadir o insertar un elemento en una lista circular varía dependiendo de la posición en que se desea insertar el elemento que inserta el nodo en la lista circular. En todo caso hay que seguir los siguientes pasos:

- Asignar memoria al nuevo nodo `nuevo` y almacenar el elemento en el atributo `e`.
- Si la lista está vacía enlazar el atributo `Sig` del nuevo nodo `nuevo` con el propio nuevo nodo, `nuevo` y poner el puntero de la lista circular en el nuevo nodo `nuevo`.
- Si la lista no está vacía se debe decidir el lugar donde colocar el nuevo nodo `nuevo`, quedándose con la dirección del nodo inmediatamente anterior `ant`. Enlazar el atributo `Sig` de nuevo nodo `nuevo` con el atributo `Sig` del nodo anterior `ant`. Enlazar el atributo `Sig` del nodo anterior `ant` con el nuevo nodo `nuevo`. Si se pretende que el nuevo nodo `nuevo` ya insertado sea el primero de la lista circular, mover el puntero de la lista circular al nuevo nodo `nuevo`. En otro caso no hacer nada.

Ejemplo 17.18

Segmento de código de una función miembro de la clase `ListaS` que inserta un elemento en una lista circular simplemente enlazada.

Para implementar una lista circular simplemente enlazada, se puede usar la clase `ListaS`, y la clase `Nodo` de las listas simplemente enlazadas. El código que se presenta corresponde a una función miembro que añade a una lista circular simplemente enlazada un elemento `e`.

```
void ListaS::Insertar(Telemento e)
{
  Nodo *ant = NULL, nuevo;
// Se busca dónde colocar el elemento x, dejando en ant la posición del
//                         nodo inmediatamente anterior si es que existe.

   nuevo = new Nodo(e);
   if (p == NULL) //la lista está vacía y sólo contiene el nuevo elemento
    {
      nuevo->PSig(nuevo);
      p = nuevo;
    }
   else            // lista no vacía, no se inserta como primer elemento.
   {
     nuevo->PSig (ant);
     ant->PSig(nuevo);
}
```

Eliminación de un elemento en una lista circular simplemente enlazada

El algoritmo para eliminar un nodo de una lista circular es el siguiente:

- Buscar el nodo `ptrnodo` que contiene el dato quedándose con un puntero al anterior `ant`.
- Se enlaza el atributo `Sig` el nodo anterior `ant` con el atributo siguiente `Sig` del nodo a borrar. Si la lista contenía un solo nodo se pone a `NULL` el atributo `p` de la lista.
- En caso de que el nodo a eliminar sea el referenciado por el puntero de acceso a la lista, `p`, y contenga más de un nodo se modifica `p` para que tenga la del atributo `Sig` de `p` (si la lista se quedara vacía hacer que `p` tome el valor `NULL`).
- Por último, se libera la memoria ocupada por el nodo.

Ejemplo 17.19

Segmento de código de una función miembro de la clase `ListaS` *que borra un elemento en una lista circular simplemente enlazada.*

```
void ListaS::Borrar(Telemento e)
{
  Nodo *ant = NULL, ptrnodo = p ;
  // búsqueda del elemento a borrar con éxito que es omitida

  ant->Psig(ptrnodo->OS());
  if (p == p->OSig())                          // la lista se queda vacía
     p = NULL;
   else if (ptrnodo == p)                      //si es el primero mover p
       p = ant->Osig();
}
```

RESUMEN

- Una lista lineal es una lista en la que cada elemento tiene un único sucesor.
- Existen cuatro operaciones típicas asociadas con listas lineales: inserción, supresión, recuperación y recorrido.
- Una **lista enlazada** es una colección ordenada de datos en los que cada elemento contiene la posición (dirección) del siguiente elemento. Es decir, cada elemento (nodo) de la lista contiene dos partes: datos y enlace (puntero).
- Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo.
- Cuando se desea insertar un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a la lista vacía, añadir al principio, añadir en el interior y añadir al final.
- Si se desea borrar un nodo de una lista se deben considerar dos casos: borrar el primer nodo y borrar cualquier otro nodo.
- El recorrido de una lista enlazada significa ir por la lista (visitar) nodo a nodo y procesar cada una.
- Una **lista doblemente enlazada** es una lista en la que cada nodo tiene un puntero a su sucesor y otro a su predecesor.
- Una **lista enlazada circularmente** es una lista en la que el enlace del último nodo apunta al primero de la lista.

EJERCICIOS

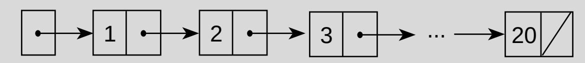
17.1. Escribir una función que calcule el número de nodos de una lista enlazada.

17.2. Escribir una función que elimine de una lista L el *n-ésimo* elemento.

17.3. Escribir una función que determine la longitud de una lista enlazada.

17.4. Escribir el código que muestra si una lista enlazada está vacía.

17.5. Escribir el segmento de código que crea la lista enlazada con los datos 1, 2 ... 20.



17.6. Escribir una función que cuente el número de veces que una determinada clave se repite en una lista secuencial.

17.7. Escribir un algoritmo que lea una lista de enteros del teclado, cree una lista enlazada con ellos e imprima el resultado de la misma.

17.8. Escribir un algoritmo que acepte una lista enlazada, la recorra y devuelva el dato del nodo con el valor menor.

17.9. Escribir un programa que intercambie dos nodos de una lista enlazada. Los nodos se identifican por número y se pasan como parámetros. Por ejemplo, para intercambiar los nodos 5 y 8 se debe llamar a `intercambio(5, 8)`. Si el intercambio se realiza con éxito, se devuelve verdadero; si se encuentra un error, tal como un número de nodo no válido, se devuelve falso.

PROBLEMAS

17.1. Escribir un programa que lea un archivo y construya una lista enlazada. Después que se construya la lista, se visualiza en el monitor. Se puede utilizar cualquier estructura de datos adecuada pero que tenga un campo clave y datos. Dos casos pueden ser: una lista de CD de música o bien una agenda de números de teléfonos.

17.2. Supongamos que se dispone de una lista enlazada en la que los datos de cada elemento contienen un entero. Escribir una función para determinar si la lista está ordenada.

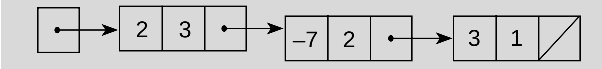
17.3. Escribir un programa que lea una lista de estudiantes de un archivo y cree una lista enlazada. Cada entrada de la lista enlazada ha de tener el nombre del estudiante, un puntero al siguiente estudiante y un puntero a una lista enlazada de calificaciones. Por ejemplo, cinco calificaciones por cada estudiante.

17.4. Escribir un programa que cree un array de listas enlazadas.

17.5. Escribir un programa que sume y reste polinomios. Cada polinomio se representa como una lista enlazada. El primer nodo de la lista representa el primer elemento del polinomio, el segundo nodo representa el segundo elemento, etcétera. Cada nodo contiene tres campos. El primer campo es el coeficiente del término; el segundo campo es la potencia del término (el exponente) y el tercer campo es un puntero al siguiente término. Ejemplo, si un término es $5x^4$ su representación en el nodo sería



y un polígono $2x^3 - 7x^2 + 3x$



Ejemplo p1: $5x^4 + 8x^3 - 9$
p2: $2x^3 - 7x^2 + 5x$
p1 + p2: $5x^4 + 10x^3 - 7x^2 + 5x - 9$

17.6. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa que permita representar mediante listas un vector disperso. A continuación realizar las operaciones:

- Suma de dos vectores dispersos.
- Producto escalar de dos vectores dispersos.

17.7. Dada una lista doblemente enlazada de números enteros, escribir el programa necesario para conseguir que dicha lista esté ordenada en orden creciente.

17.8. Escribir una función que tome una lista enlazada de enteros e invierta el orden de sus nodos.

17.9. Se dispone de una lista doblemente enlazada ordenada con claves repetidas. Realizar una función de inserción de una clave en la lista de forma tal que si la clave ya se encuentra en la lista se inserte al final de todas las que tienen la misma clave.

17.10. En una lista simplemente enlazada `L` se encuentran nombres de personas ordenadas alfabéticamente. A partir de dicha lista `L` crear una lista doblemente enlazada `LL` de tal forma que el puntero de inicio de la lista esté apuntando a la posición central. Se da por supuesto que la posición central es el nodo que ocupa la posición $n/2$ siendo n el número de nodos de la lista.

17.11. Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto así como su frecuencia de aparición.

17.12. Utilizar una lista enlazada para controlar una lista de pasajeros de una línea aérea. El programa principal debe ser controlado por menú y permitir al usuario visualizar los datos de un pasajero determinado, visualizar la lista completa, crear una lista, insertar un nodo, borrar un nodo y sustituir los datos personales de un determinado pasajero.

17.13. Diseñar una función que inserte nodos en una lista enlazada y en una posición determinada. Repetir la operación para el caso de una lista doblemente enlazada.

17.14. Construir un programa que gestione una lista doblemente enlazada que permanezca ordenada durante la ejecución del programa.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 339).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

17.1. Escribir una función miembro de una clase lista simplemente enlazada que devuelva «verdadero» si está vacía y falso «false» en otro caso; y otra función miembro que cree una lista vacía.

17.2. Escribir una función miembro que devuelva el número de nodos de una lista simplemente enlazada.

17.3. Escribir una función miembro de una clase lista simplemente enlazada que borre el primer nodo.

17.4. Escribir una función miembro de una clase lista simplemente enlazada que muestre los elementos en el orden que se encuentran.

17.5. Escribir una función miembro de una clase lista simplemente enlazada que devuelva el primer elemento de la clase lista si es que lo tiene.

17.6. Escribir una función miembro de una clase lista doblemente enlazada que añada un elemento como primer nodo de la lista.

17.7. Escribir una función miembro de una clase lista doblemente enlazada que devuelva el primer elemento de la lista.

17.8. Escribir una función miembro de una clase lista doblemente enlazada que elimine el primer elemento de la lista si es que lo tiene.

17.9. Escribir una función miembro de la clase lista doblemente enlazada que decida si está vacía.

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 342).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

17.1. Escribir el código de un constructor de copia de una clase lista simplemente enlazada.

17.2. Escribir el código de un destructor que elimine todos los nodos de un objeto de la clase lista simplemente enlazada.

17.3. Escribir una función miembro de la clase lista simplemente enlazada, que inserte en una lista enlazada ordenada crecientemente, un elemento x.

17.4. Escribir una función miembro de una lista simplemente enlazada, que elimine todas las apariciones de un elemento x que reciba como parámetro.

17.5. Escriba un método de la clase lista simplemente enlazada que elimine un elemento x que reciba como parámetro de una lista enlazada ordenada.

17.6. Escribir una función miembro de una clase lista simplemente enlazada que ordene la lista moviendo solamente punteros.

17.7. Escribir una función que no sea miembro de la clase lista simplemente enlazada, que reciba como parámetro dos objetos de tipo lista que contengan dos listas ordenadas crecientemente, y las mezcle en otra lista que reciba como parámetro.

17.8. Escribir el código de un constructor de copia de una clase lista doblemente enlazada.

17.9. Escribir una función miembro de una clase lista doblemente enlazada que inserte un elemento x en la lista y que quede ordenada.

17.10. Escribir una función miembro de una clase lista doblemente enlazada que borre un elemento x en la lista.

17.11. Escribir una función que no sea miembro de la clase lista doblemente enlazada que copie una lista doble en otra.

17.12. Escribir una clase lista simplemente enlazada circular que permita, decidir si la lista está vacía, ver cuál es el primero de la lista, añadir un elemento como primero de la lista, borrar el primer elemento de la lista y borrar la primera aparición de un elemento x de la lista.

CAPÍTULO 18

Pilas y colas

Contenido

INTRODUCCIÓN

En este capítulo se estudian en detalle las estructuras de datos pilas y colas que son probablemente las utilizadas más frecuentemente en los programas más usuales. Son estructuras de datos que almacenan y recuperan sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras **LIFO** (*Last-in, first-out*, último en entrar-primero en salir) y las colas como estructuras **FIFO** (*First-in, First-out*, primero en entrar-primero en salir).

CONCEPTOS CLAVE

- Clase Cola.
- Clase Pila.
- Cola.
- FIFO.
- LIFO.
- Pila.

18.1. CONCEPTO DE PILA

Una **pila** (*stack*) es una colección ordenada de elementos a los que sólo se puede acceder por un único lugar o extremo de la pila. Los elementos de la pila se añaden o se quitan (borran) de la misma sólo por su parte superior (**cima**). Éste es el caso de una pila de platos, una pila de libros, etc.

> Una pila es una estructura de datos de entradas ordenadas de forma que sólo se pueden introducir y eliminar por un extremo, llamado **cima**.

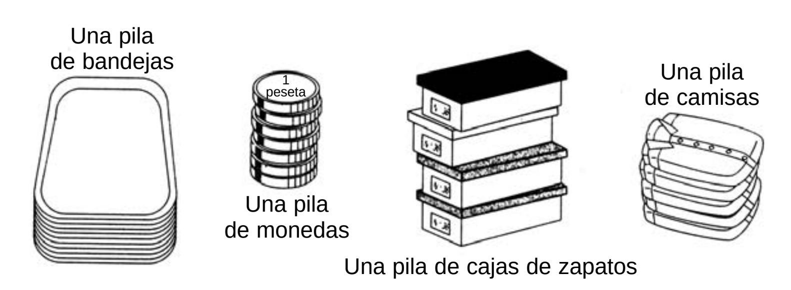


Figura 18.1. Diferentes tipos de pilas de objetos.

Cuando se dice que la pila está ordenada, lo que se quiere decir es que hay un elemento al que se puede acceder primero (el que está encima de la pila), otro elemento al que se puede acceder en segundo lugar (justo el elemento que está debajo de la cima), un tercero, etc. No se requiere que las entradas se puedan comparar utilizando el operador «menor que» (<) y pueden ser de cualquier tipo.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela de modo que la pila tendrá la novela en la parte superior.

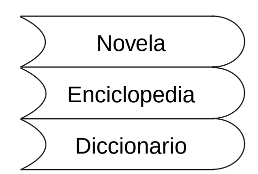


Figura 18.2. Pila de libros.

Cuando se quitan los libros de la pila, primero debe quitarse la novela, luego la enciclopedia y, por último, el diccionario.

Debido a su propiedad específica «*último en entrar-primero en salir*» se conoce a las pilas como estructura de datos **LIFO** (*last-in, first-out*).

Las operaciones usuales en la pila son *Insertar* y *Quitar*. La operación **`Insertar`** (`push`) añade un elemento en la cima de la pila y la operación **`Quitar`** (`pop`) elimina o saca un elemento de la pila. La Figura 18.3 muestra una secuencia de operaciones `Insertar` y `Quitar`. El último elemento añadido a la pila es el primero que se quita de ésta.

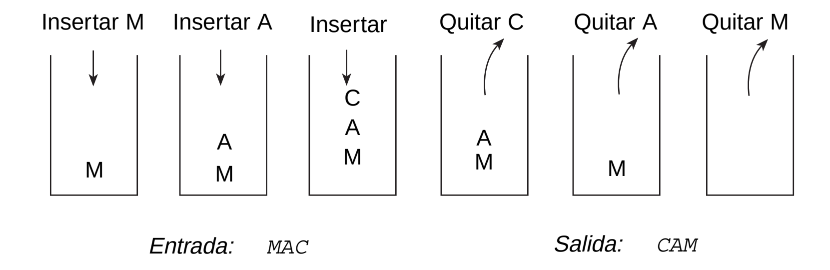


Figura 18.3. Poner y quitar elementos de la pila.

La operación `Insertar` (*push*) sitúa un elemento dato en la cima de la pila y `Quitar` (*pop*) elimina o quita el elemento de la pila.

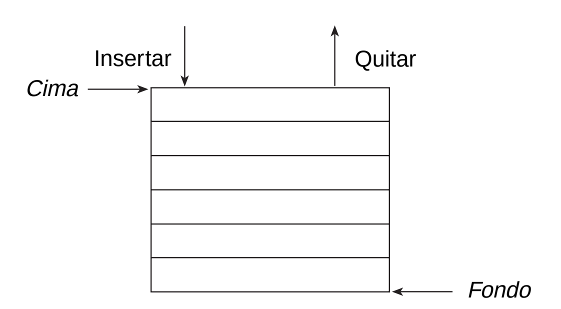


Figura 18.4. Operaciones básicas de una pila.

La pila se puede implementar mediante arrays en cuyo caso su dimensión o longitud es fija, y mediante punteros o listas enlazadas en cuyo caso se utiliza memoria dinámica y no existe limitación en su tamaño.

Una pila puede estar *vacía* (no tiene elementos) o *llena* (en el caso de tener tamaño fijo, si no caben más elementos en la pila). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** (*underflow*). Por el contrario, si un programa intenta poner un elemento en una pila, se produce un error llamado **desbordamiento** (*overflow*) o *rebosamiento*. Pata evitar estas situaciones se diseñan funciones, que comprueban si la pila está llena o vacía.

18.1.1. Especificación de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes (no todas ellas se implementan al definir una pila):

Tipo de dato	Dato que se almacena en la pila
Insertar (push)	Insertar un dato en la pila
Quitar (pop)	Sacar (quitar) un dato de la pila
Pila vacía	Comprobar si la pila no tiene elementos
Pila llena	Comprobar si la pila está llena de elementos
Limpiar pila	Quitar todos sus elementos y dejarla vacía
Tamaño de la pila	Número de elementos máximo que puede contener la pila

18.2. LA *CLASE PILA* IMPLEMENTADA CON ARRAYS

Una pila se puede implementar mediante arrays o mediante listas enlazadas. Una implementación estática se realiza utilizando un array de tamaño fijo y una implementación dinámica mediante una lista enlazada. Otro método de diseño y construcción de pilas puede realizarse mediante plantillas con la biblioteca STL. El mejor sistema para definir una pila es con una clase (`class`) o bien con una plantilla (`template`).

Los miembros de una pila incluyen una lista, un índice o puntero a la cima de la pila y el conjunto de operaciones de la pila. Se utiliza un array para contener los elementos de la pila. Un primer resultado a tener en cuenta es que el tamaño de la pila no puede exceder el número de elementos del array y la condición *pila llena* será significativa para el diseño.

El método usual de introducir elementos en una pila es definir el *fondo* de la pila en la posición 0 del array y sin ningún elemento en su interior, es decir, definir una *pila vacía*; a continuación, se van introduciendo elementos en el array (en la pila) de modo que el primer elemento añadido se introduce en una pila vacía y en la posición 0, el segundo elemento en la posición 1, el siguiente en la posición 2, y así sucesivamente. Con estas operaciones el puntero (apuntador) que señala a la cima de la pila se va incrementando en 1 cada vez que se añade un nuevo elemento; es decir, el *puntero de la pila* almacena el índice del array que se está utilizando como cima de la pila. Los algoritmos de introducir «insertar» (*push*) y quitar «sacar» (*pop*) datos de la pila utilizando el índice del array como puntero de la pila son:

```
Insertar (push)

1. Verificar si la pila no está llena.
2. Incrementar en 1 el puntero de la pila.
3. Almacenar elemento en la posición del puntero de la pila.

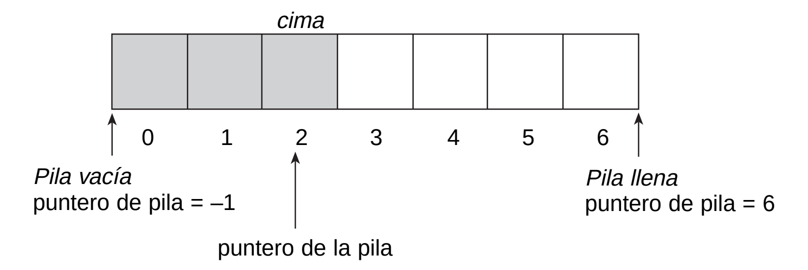
Quitar (pop)

4. Si la pila no está vacía.
5. Leer el elemento de la posición del puntero de la pila.
6. Decrementar en 1 el puntero de la pila.
```

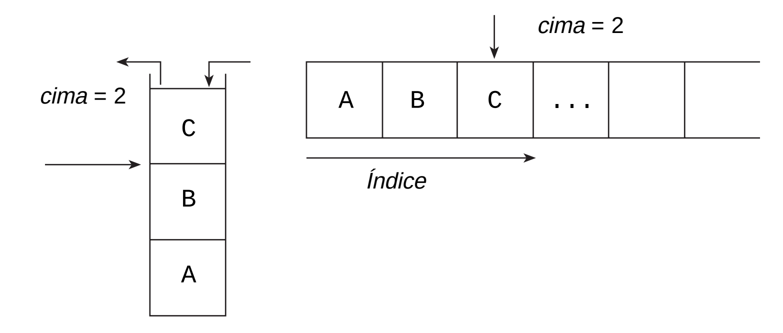
En el caso de que el array que define la pila tenga `TamanioPila` elementos, las posiciones del array, es decir el índice o puntero de la pila, estarán comprendidas en el rango `0` a `TamanioPila-1` elementos, de modo que *en una pila llena* el puntero de la pila apunta a `TamanioPila-1` y *en una pila vacía* el puntero de la pila apunta a –1, ya que 0, teóricamente, será el índice del primer elemento.

Ejemplo 18.1

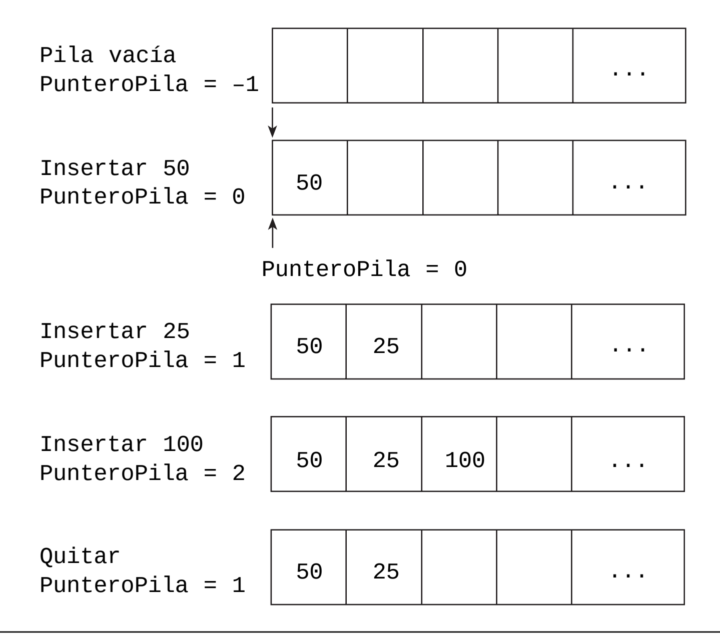
Una pila de 7 elementos se puede representar gráficamente así:



Si se almacenan los datos `A, B, C`, ... en la pila se puede representar gráficamente por alguno de estos métodos:



Veamos ahora cómo queda la pila en función de diferentes situaciones de un posible programa.



18.2.1. Especificación de la clase pila

La declaración de una pila incluye los datos y operaciones ya citados anteriormente.

1. Datos de la pila (tipo `TipoData`, que es conveniente definirlo mediante `typedef`).
2. Verificar que la pila no está llena antes de intentar insertar o poner (*«push»*) un elemento en la pila; verificar que una pila no está vacía antes de intentar *quitar* o *sacar* (*«pop»*) un elemento de la pila. Si estas precondiciones no se cumplen se debe visualizar un mensaje de error y el programa debe terminar.
3. `Pilavacía` devuelve 1 (verdadero) si la pila está vacía y 0 (falso) en caso contrario.
4. `Pilallena` devuelve 1 (verdadero) si la pila está llena y 0 (falso) en caso contrario. Estas funciones se utilizan para verificar las operaciones del párrafo 2.
5. `LimpiarPila`. Se limpia o vacía la pila, dejándola sin elementos y disponible para otras tareas.
6. `VerPila` devuelve el valor situado en la cima de la pila, pero no se decrementa el puntero de la pila, ya que ésta queda intacta.

Declaración

```
// archivo pilaarray.h
#include <iostream>
#include <stdlib.h>
using namespace std;

const int MaxTamaPila = 100;

class Pila
{
   private:
      // array de la pila y cima
      TipoDato listapila[MaxTamaPila];
      int cima;
   public:
      // constructor, inicializa cima o puntero de pila
      Pila(void);
      // operaciones de la pila
      void Insertar(const TipoDato& elemento);
      TipoDato Quitar(void);
      void LimpiarPila(void);

      // acceso a pila
      TipoDato VerPila(void) const;

      // verificación estado de la pila
      int PilaVacia(void) const;
      int PilaLlena(void) const;
};
```

Ejemplo 18.2

Escribir un programa que manipule la clase `Pila` *definida anteriormente e introduzca un dato de tipo entero tal como 25.*

```
typedef int TipoDato;
#include "pilaarray.h";          // incluye la clase Pila
using namespace std;

Pila P;

P. Insertar(25);                  // insertar 25 en la pila P
cout << P.VerPila() << endl;   // visualizar 25

// Quita el elemento 25 y deja la pila vacía
if (!P.PilaVacia())
    aux = P.Quitar();
cout << aux << endl;

P.LimpiarPila();                  // limpiar pila
```

18.2.2. Implementación

Las operaciones de la pila definidas en la especificación y el constructor se pueden implementar así:

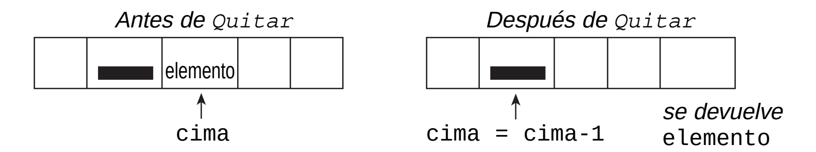
```
// operación constructor
// inicializar cima de la pila
Pila::Pila(void) : cima(-1)
{}
```

Las operaciones de la pila definidas en la clase `Pila` son: `Insertar`, `Quitar` y `VerPila`. Las operaciones `Insertar` y `Quitar` insertan y eliminan un elemento de la pila; la operación `VerPila` permite a un cliente recuperar los datos de la cima de la pila sin quitar realmente el elemento de la misma.

La operación `Insertar` un elemento en la pila incrementa el puntero de la pila (cima) en 1 y asigna el nuevo elemento a la lista de la pila. Cualquier intento de añadir un elemento en una pila llena produce un mensaje de error `"Desbordamiento pila"` y debe terminar el programa.

```
// poner un elemento en la pila
void Pila::Insertar (const TipoDato& elemento)
{
   //si la pila está llena, terminar el programa
   if (cima == MaxTamPila-1)
   {
      cerr << "Desbordamiento pila" << endl;
      exit (1);
   }
   // incrementar puntero pila y copiar elemento en listapila
   cima++;
   listapila[cima] = elemento;
}
```

La operación `Quitar` elimina un elemento de la pila copiando primero el valor de la cima de la pila en una variable local auxiliar y, a continuación, decrementa el puntero de la pila en 1. La variable `aux` se devuelve en la ejecución de la operación `Quitar`. Si se intenta eliminar o borrar un elemento en una pila vacía se debe producir un mensaje de error y el programa debe terminar.




```
// Quitar un elemento de la pila
TipoDato Pila::Quitar (void)
{
   Tipodato Aux;
   // si la pila está vacía, terminar el programa
   if (cima == -1)
   {
      cerr << "Se intenta sacar un elemento en pila vacía" << endl;
      exit (1);
   }
```

```
      // guardar elemento de la cima
      aux = listapila[cima];

      // decrementar cima y devolver valor del elemento
      cima--;
      return aux;
   }
```

18.2.3. Operaciones de verificación del estado de la pila

Se debe proteger la integridad de la pila, para lo cual la clase `Pila` ha de proporcionar operaciones que comprueben el estado de la pila : *pila vacía* o *pila llena.* Asimismo, se ha de definir una operación que restaure la condición inicial de la pila, que fue determinada por el constructor (cima de la pila a –1), `LimpiarPila.`

La función `PilaVacia` comprueba (verifica) si la cima de la pila es –1. En ese caso, la pila está vacía y se devuelve un 1 (verdadero); en caso contrario, se devuelve 0 (falso).

```
// verificar pila vacia
int Pila::PilaVacia(void) const
{
   // devuelve el valor lógico resultante de expresión cima == -1
   return cima == -1;
}
```

La función `PilaLlena` comprueba (verifica) si la cima es `MaxTamPila-1`. En ese caso, la pila está llena y se devuelve un 1 (verdadero); en caso contrario, se devuelve 0 (falso).

```
// verificar si la pila está llena
int Pila::PilaLlena (void) const
{
   // devuelve la posición de la cima
   return cima == MaxTamPila-1;
}
```

Por último, la operación `LimpiarPila` reinicializa la cima a su valor inicial con la pila vacía (–1)

```
// quitar todos los elementos de la pila
void Pila::LimpiarPila (void)
{
   cima = -1;
}
```

Ejemplo 18.3

Clase pila implementada con array.

Se define una constante `MaxTamaPila` cuyo valor es 100 que será el valor máximo de elementos que podrá contener la pila. Se define en un `typedef` el tipo de datos que almacenará `Pila`, que en este caso serán enteros. `Pila` es una clase cuyos atributos son la `cima` que apunta siempre al último elemento añadido a la pila y un array `A` cuyos índices variarán entre `0` y `MaxTamaPila–1`. Todas la funciones miembro de la clase, son públicas, excepto `EstallenaP` que es privada.

- `VaciaP`. Crea la pila vacía poniendo la cima en el valor `–1`.
- `Pila`. Es el constructor de la `Pila`. Coincide con `VaciaP`.
- `EsvaciaP`. Decide si la pila vacía. Ocurre cuando su `cima` valga `–1`.
- `EstallenaP`. En la implementación de una pila con array, hay que tenerla declarada como privada para prevenir posibles errores. En este caso la pila estará llena cuando la cima apunte al valor `MaxTamaPila–1`.
- `AnadeP`. Añade un elemento a la pila. Para hacerlo comprueba en primer lugar que la pila no esté llena, y en caso afirmativo, incrementa la `cima` en una unidad, para posteriormente poner en el array `A` en la posición `cima` el elemento.
- `PrimeroP`. Comprueba que la pila no esté vacía, y en caso de que así sea dará el elemento del array `A` almacenado en la posición apuntada por la `cima`.
- `BorrarP`. Se encarga de eliminar el último elemento que entró en la pila. Primeramente, comprueba que la pila no esté vacía en cuyo caso, disminuye la `cima` en una unidad.
- `Pop`. Esta operación extrae el primer elemento de la pila y lo borra. Puede ser implementada directamente, o bien llamando a las primitivas `PrimeroP` y posteriormente a `BorrarP`.
- `Push`. Esta primitiva coincide con `AnadeP`.

```
#include <cstdlib>
#include <iostream>
using namespace std;

#define MaxTamaPila 100

typedef int TipoDato;
class Pila
{
  protected:
     TipoDato A[MaxTamaPila];
     int cima;
  public:
     Pila(){ cima = -1;}                                    //constructor
     ~Pila(){}                                               //destructor
     void VaciaP();
     void AnadeP(TipoDato elemento);
     void BorrarP();
     TipoDato PrimeroP();
     bool EsVaciaP();
     void Push(TipoDato elemento);
     TipoDato Pop();
  private:
     bool EstallenaP();
};

void Pila:: VaciaP()
{
     cima = -1;
}
void Pila::AnadeP(TipoDato elemento)
{
     if (EstallenaP())
     {
       cout << " Desbordamiento pila";
```

```
          exit (1);
        }
        cima++;
        A[cima] = elemento;
}
void Pila::Push (TipoDato elemento)
{
        AnadeP (elemento);
}
TipoDato Pila::Pop()
{
  TipoDato Aux;
  if (EsVaciaP())
  {
    cout <<"Se intenta sacar un elemento en pila vacía";
    exit (1);
  }
   Aux = A[cima];
   cima--;
   return Aux;
}

TipoDato Pila::PrimeroP()
{
  if (EsVaciaP())
  {
      cout <<"Se intenta sacar un elemento en pila vacía";
      exit (1);
  }
  return A[cima];
}

void Pila::BorrarP()
{
  if (EsVaciaP())
  {
      cout <<"Se intenta sacar un elemento en pila vacía";
      exit (1);
  }
  cima--;
}
bool Pila::EsVaciaP()
{
        return cima == -1;
}
bool Pila::EstallenaP()
{
        return cima == MaxTamaPila-1;
}

int main(int argc, char *argv[])
{
```

```
        Pila P;

        P.AnadeP(5);
        P.AnadeP(6);
        cout << P.Pop() << endl;;
        cout << P.Pop() << endl;
        system("PAUSE");
        return EXIT_SUCCESS;
}
```

18.2.4. La clase pila implementada con punteros

Para implementar una pila con punteros basta con usar una lista simplemente enlazada, con lo que la pila estará vacía si la lista apunta a `NULL`. La pila teóricamente nunca estará llena. La pila se declara como una clase que tiene un atributo protegido que es un puntero a nodo, este puntero señala el extremo de la lista enlazada por el que se efectúan las operaciones. Siempre que se quiera poner un elemento se hace por el mismo extremo que se extrae.

Los algoritmos de introducir «insertar» (*push*) y quitar «sacar» (*pop*) datos de la pila son:

Insertar (*push*). Basta con añadir un nuevo nodo con el dato que se quiere insertar como primer elemento de la lista (pila).

Quitar (*pop*). Verificar si la lista (pila) no está vacía. Extraer el valor del primer nodo de la lista (pila). Borrar el primer nodo de la lista (pila).

Ejemplo 18.4

Clase `Pila` implementada con punteros.

Se define en primer lugar la clase `Nodo`, ya implementada en el Ejercicio resuelto 18.3 y utilizada en las listas simplemente enlazadas. La clase `Pila` tiene por atributo protegido un puntero a la clase `Nodo`. Las funciones miembros que se implementan de la clase `Pila` son:

- Constructor `Pila`. Crea la pila vacía poniendo el atributo `p` a `NULL`.
- Constructor de copia `Pila` utilizado para la transmisión de parámetros por valor de una pila a una función.
- Destructor `~Pila`, cuyo código elimina todos los nodos de la lista.
- `VaciaP`. Crea la pila vacía poniendo el atributo `p` a `NULL`.
- `EsvaciaP`. Decide si la pila está vacía. Esto ocurre cuando el atributo `p` valga `NULL`.
- `AnadeP`. Añade un elemento a la pila. Para hacerlo es preciso añadir un nuevo nodo que contenga como información el elemento que se quiera añadir y ponerlo como primero de la lista enlazada.
- `PrimeroP`. En primer lugar se comprobará que la pila (lista) no esté vacía, y en caso de que así sea, dará el campo almacenado en el primer nodo de la lista enlazada.
- `BorrarP`. Se encarga de eliminar el último elemento que entró en la pila. Primeramente comprueba que la pila no esté vacía, en cuyo caso, se borra el primer nodo de la pila (lista enlazada).
- `Pop`. Esta operación extrae el primer elemento de la pila y lo borra. Puede ser implementada directamente, o bien llamando a las primitivas `PrimeroP` y, posteriormente, a `BorrarP`.
- `Push`. Esta primitiva coincide con `AnadeP`.

```
#include <cstdlib>
#include <iostream>
```

```
using namespace std;

typedef int Telemento;

class Nodo {
    protected:
        Telemento e;
        Nodo *Sig;
    public:
        Nodo(){}
        ~Nodo(){}
        Nodo (Telemento x){ e = x; Sig = NULL;}         // constructor
        Telemento OE(){ return e;}                 // Obtener elemento
        void PE(Telemento x){ e = x;}                // poner elemento
        Nodo * OSig(){ return Sig;}               // Obtener siguiente
        void PSig( Nodo *p){ Sig = p;}              // poner siguiente
    };
class Pila
 {
    protected:
         Nodo *p;

    public:
        Pila(){p = NULL;}
        void VaciaP(){ p = NULL;}
        bool EsvaciaP(){ return !p;}
        Telemento PrimeroP(){ if (p) return p->OE();}
        void AnadeP( Telemento e);
        void Push(Telemento e){ AnadeP(e);}
        void BorrarP();
        Pila::Pila (const Pila &p2);
        ~Pila();
        Telemento Pop();
 };

void Pila::AnadeP(Telemento e)
{
  Nodo *aux;

  aux = new Nodo(e);
  if( p )
   aux->PSig(p);
  p = aux;
}

void Pila::BorrarP()
{
   Nodo *paux;

   if(p)
    {
      paux = p;
```

```
      p = p->OSig();
      delete paux;
   }
   else;            // error
}

Pila::Pila (const Pila &p2)
{
   Nodo* a = p2.p, *af, *aux;

   p = NULL;
   AnadeP(-1);                                  // Se añade Nodo Cabecera
   af = p;
   while ( a != NULL)
   {
       aux = new Nodo(a->OE());
       af->PSig(aux);
       af = aux;
       a = a->OSig();
   }
   BorrarP();                                   // Se borra nodo Cabecera

}

Pila:: ~Pila()
{
   Nodo *paux;

   while(p != 0)
   {
       paux = p;
       p = p->OSig();
       delete paux;
   }
}

Telemento Pila::Pop()
{
  Telemento e;

  e = PrimeroP();
  BorrarP();
  return e;
}

int main(int argc, char *argv[])
{
  Pila p;

  p.VaciaP();
  p.AnadeP(4);
```

```
    p.AnadeP(5);
    while (!p.EsvaciaP())
    {
      cout << p.PrimeroP() << endl;
      p.BorrarP();
    }
     system("PAUSE");
     return EXIT_SUCCESS;
}
```

18.3. COLAS

Una **cola** es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista (Fig. 18.5). Un elemento se inserta en la cola (parte final) de la lista y se suprime o elimina por la frente (parte inicial, cabeza o frente) de la lista. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.

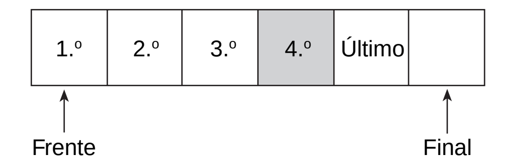


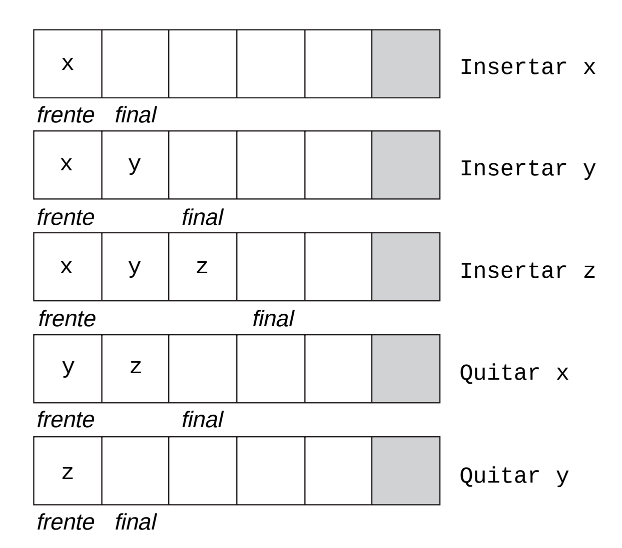
Figura 18.5. Una cola.

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan y, por consiguiente, una cola es una estructura de tipo **FIFO** (*first-in/firs-out, primero en entrar-primero en salir* o bien *primero en llegar-primero en ser servido*). El servicio de atención a clientes en un almacén es un ejemplo típico de cola. La acción de gestión de memoria intermedia (*buffering*) de trabajos o tareas de impresora en un distribuidor de impresoras (*spooler*) es otro ejemplo típico de cola[1]. Dado que la impresión es una tarea (un trabajo) que requiere más tiempo que el proceso de la transmisión real de los datos desde la computadora a la impresora, se organiza una cola de trabajos de modo que los trabajos se imprimen en el mismo orden en que se recibieron por la impresora. Este sistema tiene el gran inconveniente de que si su trabajo personal consta de una única página para imprimir y delante de su petición de impresión existe otra petición para imprimir un informe de 300 páginas, deberá esperar a la impresión de esas páginas antes de que se imprima la suya.

Desde el punto de vista de estructura de datos, una cola es similar a una pila, en donde los datos se almacenan de un modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola. Las *acciones* que están *permitidas* en una cola son:

- Creación de una cola vacía.
- Verificación de que una cola está vacía.
- Añadir un dato al final de una cola.
- Eliminación de los datos de la cabeza de la cola.

[1] Recordemos que este caso sucede en sistemas multiusuario donde hay varios terminales y sólo una impresora de servicio. Los trabajos se «encolan» en la cola de impresión.



18.3.1. La clase cola implementada con arrays

Al igual que las pilas, las colas se pueden implementar utilizando arrays o listas enlazadas. En esta sección consideraremos la implementación utilizando arrays.

La definición de una clase `Cola` que contiene un array para almacenar los elementos de la cola, y dos marcadores o punteros (variables) que mantienen las posiciones frente y final de la cola; es decir, un marcador apuntando a la posición de la cabeza de la cola y el otro al primer espacio vacío que sigue al final de la cola. Cuando un elemento se añade a la cola, se verifica si el marcador final apunta a una posición válida, entonces se añade el elemento a la cola y se incrementa el marcador final en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador (puntero) de cabeza y éste se incrementa en 1.

Este procedimiento funciona bien hasta la primera vez que el puntero de cabeza o cabecera alcanza el extremo del array y éste queda o bien vacío o bien lleno.

Definición de la especificación de una `cola`

Una cola debe manejar diferentes tipos de datos; por esta circunstancia, se define en primer lugar el tipo genérico `TipoDato`. La clase `Cola` contiene una lista (`listaQ`) cuyo máximo tamaño se determina por la constante `MaxTamQ`.

Se definen tres tipos de variables puntero o marcadores, `frente,  final` y `cuenta`. `frente` y `final` son los punteros de cabecera y cola o final. `cuenta` es una variable que lleva el control del número de elementos de la cola y que sirve para determinar si la cola está vacía o llena.

Las operaciones típicas de la cola son: `InsertarQ, EliminarQ, Qvacia, Qllena`, y `FrenteQ`. `InsertarQ` toma un elemento del tipo `TipoDato` y lo inserta en el final de la cola. `EliminarQ` elimina (quita) y devuelve el elemento de la cabeza o frente de la cola. El método `FrenteQ` devuelve el valor del elemento en el frente de la cola, que permite «visualizar» al elemento siguiente que se eliminará.

La operación `Qvacia` comprueba si la cola está vacía antes de eliminar un elemento y `Qllena` comprueba si la cola está llena antes de insertar un nuevo miembro. Si las precondiciones para `InsertarQ` y `EliminarQ` se violan, el programa debe imprimir un mensaje de error y terminar.

Especificación de la clase `cola`

```
#include <iostream>
#include <stdlib>
using namespace std;

const int MaxTamQ = 100;
class Cola
{
   private:
      // definición del array de la cola, punteros y contador
      int frente, final, cuenta;
      TipoDato listaQ[MaxTamQ];

   public:
      Cola(void);      // inicialización de miembros dato

               // operaciones de modificación de la cola
      void InsertarQ(const TipoDato& elemento);
      TipoDato EliminarQ(void);
      void BorrarCola(void);

      // acceso a la cola
      TipoDato FrenteQ(void) const;

      // métodos de verificación del estado de la cola
      int LongitudQ(void) const;
      int Qvacia(void) const;
      int Qllena(void) const;
};
```

La declaración e implementación de la cola se almacena en un archivo de cabecera `"colaary.h"`.

Implementación de la clase `cola`

La definición de una clase que contiene un array para el almacenamiento de los elementos de la cola y dos marcadores o punteros: uno apuntando a la posición de la cabeza o cabecera de la cola y la otra al primer espacio vacío a continuación del final de la cola. Cuando un elemento se añade a la cola, se hace un test (prueba) para ver si el marcador final apunta a una posición válida, a continuación se añade el elemento a la cola y el marcador final se incrementa en 1. Cuando se quita (elimina) un elemento de la cola, se realiza un test (prueba) para ver si la cola está vacía, y si no es así, se recupera el elemento que se encuentra en la posición apuntada por el marcador de cabeza y el marcador de cabeza se incrementa en 1.

Este procedimiento funciona bien hasta la primera vez que el marcador `final` alcanza el final del array. Si durante este tiempo se han producido eliminaciones, habrá espacio vacío al principio del array. Sin embargo, puesto que el marcador final apunta al extremo del array, implicará que la cola está llena y ningún dato más se añadirá. Se pueden desplazar los datos de modo que la cabeza de la cola vuelve al principio del array cada vez que esto sucede, pero el desplazamiento de datos es costoso en términos de tiempo de computadora, especialmente si los datos almacenados en el array son objetos de datos grandes.

El medio más eficiente, sin embargo, para almacenar una cola en un array, es utilizar un tipo especial de array que una el extremo final de la cola con su extremo cabeza. Este array se denomina *circular* y

permite que el array completo se utilizará para almacenar elementos de la cola sin necesidad de que ningún dato se desplace. Un array circular con *n* elementos se visualiza en la Figura 18.6.

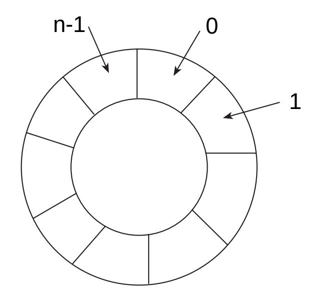


Figura 18.6. Un array circular.

El array se almacena de modo natural en la memoria tal como un bloque lineal de *n* elementos. Se necesitan dos marcadores (punteros) *cabeza* y *final* para indicar la posición de la cabeza y la posición inmediatamente después del final, donde se almacena el elemento a añadir. Una cola vacía se representa por la condición `cabeza = final`.

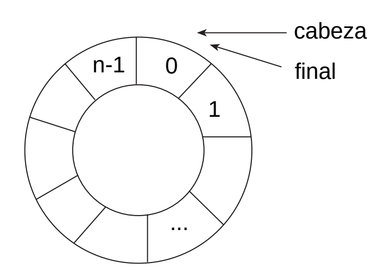


Figura 18.7. Una cola vacía.

La variable `frente` o `cabeza` es siempre la posición del primer elemento de la cola y se avanza en el sentido de las agujas del reloj. La variable `final` es la posición en donde se produce la siguiente inserción. Después de que se ha producido una inserción, `final` se mueve circularmente a la derecha. Una variable `cuenta` mantiene un registro del número de elementos, y si `cuenta` es igual a elementos `MaxTamQ`, la cola está llena.

La implementación del movimiento circular utilizando la *teoría de los restos*:

Mover final adelante = `(final + 1) % MaxTamQ`
Mover cabeza adelante = `(frente + 1) % MaxTamQ`

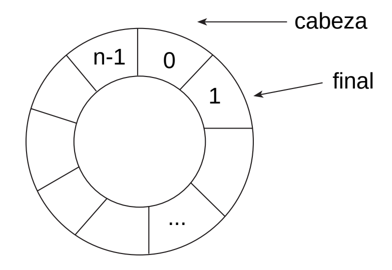


Figura 18.8. Una cola que contiene un elemento.

Los algoritmos que formalizan la gestión de colas en un array circular han de incluir al menos las siguientes tareas:

- Creación de una cola vacía: `cabeza = final = 0`.
- Comprobar si una cola está vacía:
  es `cabeza == final ?`
- Comprobar si una cola está llena:
  `(final + 1) % n == cabeza ?`
- Añadir un elemento a la cola: si la cola no está llena, añadir un elemento en la posición `final` y se establece `final = (final + 1) % n` (% operador resto).
- Eliminación de un elemento de una cola: si la cola no está vacía, eliminarlo de la posición `cabeza` y establecer `cabeza = (cabeza + 1) % n.`

Operaciones de la `cola`

Una cola permite un conjunto limitado de operaciones que añade un nuevo elemento (`Qinsertar`) o quita/elimina un elemento (`EliminarQ`). La clase `Cola` proporciona también `frenteQ`, que permite «ver» el primer elemento de la cola.

InsertarQ

Antes de que comience el proceso de inserción, el índice final apunta a la siguiente posición disponible en la lista. El nuevo elemento se sitúa en su posición y la cuenta de la cola se incrementa en 1.

```
ListaQ[final]= elemento;
cuenta++;
```

Después de situar el elemento de la lista, el índice `final` se debe actualizar para apuntar a la siguiente posición. El cálculo de las posiciones sucesivas se consigue mediante el operador resto (%).

```
// insertar elemento en la cola
void Cola::InsertarQ (const TipoDato& elemento)
{
   // terminar si la cola está llena
   if (cuenta == MaxTamQ)
   {
      cerr << "desbordamiento cola" << endl;
      exit (1);
   }
   // incrementar cuenta, asignar elemento a listaQ y actualizar final
   cuenta++;
   listaQ[final] = elemento;
   final = (final + 1)% MaxTamQ;
}
```

EliminarQ

La operación `EliminarQ` borra o elimina un elemento del frente de la cola, una posición que se referencia por el índice `frente`. Comienza el proceso de eliminación copiando el valor en una variable temporal y decrementando la cuenta de cola.

```
elemento = listaQ[frente]
cuenta--:
```

En el modelo circular, la cabeza se debe volver a posicionar en el siguiente elemento de la lista utilizando el operador resto (%).

```
frente = (frente + 1)% MaxTamQ;
```

El código fuente es :

```
// borrar elemento del frente de la cola y devuelve su valor
TipoDato Cola::EliminarQ (void)
{
   TipoDato aux;

   //si listaQ está vacía, terminar el programa
   if (cuenta == 0)
   {
      cerr << "Eliminación de una cola vacía" << endl;
         exit (1);
   }

   // registrar valor en el frente de la cola
   aux = listaQ[frente] ;

   // decrementar cuenta, avanzar frente y devolver primero
   // del frente
   cuenta--;
   frente = (frente + 1) % MaxTamQ;
   return aux ;
}
```

Ejemplo 18.5

Implementación de la clase Cola con un array circular.

La clase `Cola` contiene los atributos protegidos `frente`, `final` y el array `A` declarado de una longitud máxima. Todas las funciones miembro serán públicas, excepto el método `EstallenaC` que será privado de la clase. En la implementación de la clase `Cola` con un array circular hay que tener en cuenta que `frente` va a apuntar siempre a una posición anterior donde se encuentra el primer elemento de la cola y `final` va a apuntar siempre a la posición donde se encuentra el último de la cola. Por tanto, la parte esencial de las tareas de gestión de una cola son:

- Creación de una cola vacía: hacer `frente = final = 0`.
- Comprobar si una cola está vacía: ¿es `frente == final`?
- Comprobar si una cola está llena: ¿es `(final+1)% MaxTamC == frente`? No se confunda con cola vacía.
- Añadir un elemento a la cola: si la cola no está llena, añadir un elemento en la posición siguiente a `final` y se establece: `final = (final+1)%MaxTamC`.
- Eliminación de un elemento de una cola: si la cola no está vacía, eliminarlo de la posición siguiente a `frente` y establecer `frente = (frente+1) % MaxTamC`.

```
#include <cstdlib>
#include <iostream>
#define MaxTamC 100
using namespace std;

typedef int TipoDato;
class Cola
{
 protected:
  int frente, final;
  TipoDato A[MaxTamC];

 public:
   Cola();                                      //constructor
   ~Cola();                                     //destructor
   void VaciaC();
   void AnadeC(TipoDato e);
   void BorrarC();
   TipoDato PrimeroC();
   bool EsVaciaC();

private:
   bool EstallenaC();
};

Cola::Cola()
{
  frente = 0;
  final = 0;
 }

Cola::~Cola(){}

void Cola::VaciaC()
{
  frente = 0;
  final = 0;
}

void Cola::AnadeC(TipoDato e)
{
  if (EstallenaC( ))
  {
    cout << "desbordamiento cola";
    exit (1);
  }
  final = (final + 1) % MaxTamC;
  A[final] = e;
}
```

```
TipoDato Cola::PrimeroC()
{
  if (EsVaciaC())
  {
    cout << "Elemento frente de una cola vacía";
    exit (1);
  }
  return (A[(frente+1) % MaxTamC]);
}

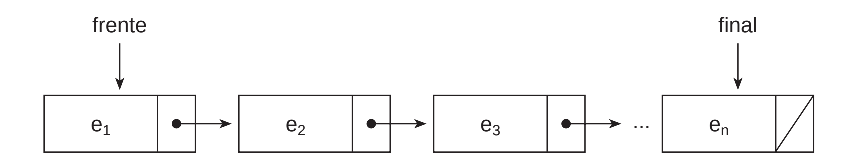
bool Cola::EsVaciaC()
{
  return (frente == final);
}

bool Cola::EstallenaC()
{
  return (frente == (final+1) % MaxTamC);
}

void Cola::BorrarC()
{
  if (EsVaciaC())
  {
    cout << "Eliminación de una cola vacía";
    exit (1);
  }
  frente = (frente + 1) % MaxTamC;
}
```

18.3.2. La clase cola implementada con una lista enlazada

Cuando la cola se implementa utilizando variables dinámicas, la memoria utilizada se ajusta en todo momento al número de elementos de la cola, pero se consume algo de memoria extra para realizar el encadenamiento entre los elementos de la cola. Se utilizan dos punteros para acceder a la cola, `frente` y `final`, que son los extremos por donde salen los elementos y por donde se insertan respectivamente.



18.4. IMPLEMENTACIÓN DE UNA PILA CON UNA LISTA ENLAZADA

Una pila se puede construir fácilmente con una simple lista enlazada. La estructura pila necesita un puntero y un nodo que contiene los datos y un puntero al siguiente nodo de la pila.

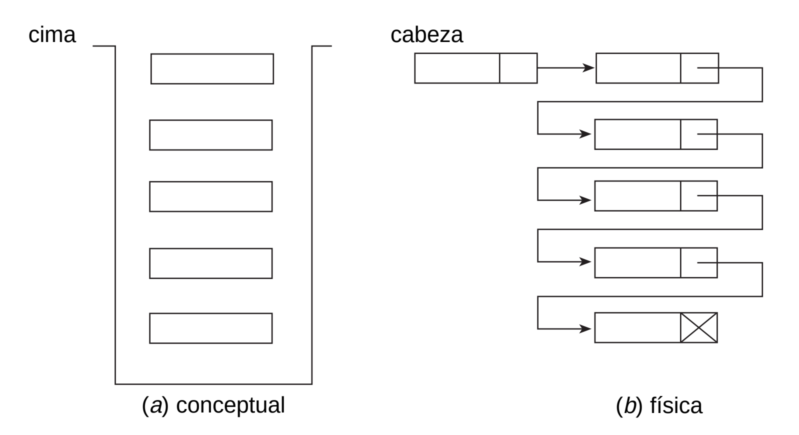


Figura 18.9. Implementaciones de una pila: (*a*) conceptual; (*b*) física.

Las operaciones que se realizan en una pila implementada con una lista enlazada son las ya conocidas:

- *Crear la pila*.
- *Insertar un elemento*.
- *Quitar un elemento*.
- *Cima de la pila*.
- *Pila vacía*.
- *Pila llena*.
- *Contar la pila*.
- *Borrar (destruir) la pila*.

La operación `Insertar` (*push*) pone un elemento en la pila (el primer elemento) y la operación `Quitar` (*pop*) elimina o suprime un elemento de la pila (el primer elemento de la lista). Se define una clase `Pila` y se pone la definición en un archivo con el nombre `pila.h`.

Ejemplo 18.6

Construir una pila con elementos enteros.

La definición de la clase `Pila`, al menos, requiere las siguientes funciones miembro/operaciones:

`Insertar`	Insertar un nuevo elemento en la lista
`Quitar`	Quita o suprime un elemento de la pila
`Vacía`	Verifica si la pila está vacía. Si se intenta extraer un elemento de una pila que está vacía, se genera una excepción de la clase estándar `length_error` (en consecuencia se declara una especificación de la excepción para representar este hecho).

Una pila tiene un único miembro dato, el puntero `Primero`, que apunta al primer elemento de una lista enlazada. El constructor por defecto inicializa este puntero a 0, que significa que cada nueva pila creada estará vacía automáticamente desde el principio. Se declara también un destructor que libera (limpia) todos los elementos de la lista asignados. La clase se almacena en el archivo `pila.h.`

```
// Archivo pila.h
#include <stdexcept>
class Elemento;
```

```
class Pila {
public:
   Pila(): primero(0) {};
   ~Pila();
   void insertar(int d);
   int quitar() throw(length_error);
   bool vacia();
private:
   Elemento *primero;
   Pila(const Pila&) {};
   Pila& operator = (const Pila&) {};
};
```

En la sección privada de la clase se incluyen dos constructores, uno de copia y un operador de asignación. Estos constructores son necesarios ya que, como conoce el lector, se define automáticamente el constructor de copia y el operador de asignación de cada clase y estos copian los miembros dato uno a uno. Si no se definiera el constructor de copia y el operador de asignación de la clase `Pila`, el puntero `Primero` se copiará cuando se copia una pila durante una inicialización o una asignación. Esto entrañaría que dos objetos pila diferentes apuntarían a la *misma lista* enlazada, lo que, naturalmente, sería incorrecto.

El archivo `pila.h` sólo contiene una *declaración* de una clase `Elemento` y no una definición completa. Sin embargo, se necesita definir la clase `Elemento` y eso se hace en el archivo `pila.cpp`, junto con las funciones miembro de la clase `Pila`:

```
// Archivo pila.cpp
class Elemento {
   friend class Pila;
   Elemento *siguiente;
   int datos;
   Elemento(Elemento *n, int d);siguiente(n), datos(d) {}
};

void Pila::insertar(int d)
{
   primero = new Elemento(primero, d);
}

int Pila::quitar() throw (length_error)
{
   if (vacia())
      throw length_error("Pila::quitar");
   Elemento *p = primero;
   int d = p -> datos;
   primero = primero -> siguiente;
   delete p;
   return d;
}

bool Pila::vacia()
{
   return !primero;
}
```

```
Pila::~Pila()
{
   while(! vacia())
   {
      Elemento *p = primero;
      primero =  primero -> siguiente;
      delete p;
   }
}
```

La clase `Pila` se ha definido como una clase amiga y, por consiguiente, tiene acceso a toda la clase `Elemento`. La función miembro `Insertar` pone un elemento en la pila. La función miembro `Quitar` es también fácil. Si la pila está vacía, se genera una excepción. En caso contrario, el primer elemento se elimina de la lista y el valor que estaba en esa posición se devuelve y se libera el espacio de memoria ocupado por el elemento. En el destructor la sentencia `while` se ejecuta hasta que la lista esté vacía.

El siguiente programa muestra cómo se utiliza la clase. El programa lee un texto del terminal y lo escribe en orden inverso (los caracteres individuales que se leen son de tipo `char` y se convierten automáticamente en datos de tipo `int` cuando se ponen en la pila).

```
#include "pila.h"
#include <iostream>
using namespace std;

void main()
{
   Pila p;
   char c;
   cout << "Escribir un texto. Terminar con Ctrl-Z"<< endl;
   while (cin.get(c))
      p. insertar(c);
   while (! p.vacia())
   {
      c = p.quitar();
      cout << c;
   }
return 0;
}
```

RESUMEN

- Una **pila** es una estructura de datos tipo **LIFO** (*last-in*, *first-out*, último en entrar, primero en salir) en la que los datos se insertan y eliminan por el mismo extremo que se denomina *cima* de la pila.
- Se definen ocho operaciones fundamentales: `crear`, `insertar`, `eliminar` (quitar), `pilaVacía`, `pilaLlena`, `contarPila` y `LiberarPila`.
- `Crear` asigna memoria a la pila. `Insertar` añade un elemento en la cima de la pila. Una operación de `Quitar`, elimina un elemento de la cima de la pila y el siguiente elemento, si existe, se queda como cima. Cada operación `Insertar` debe asegurarse de que existe espacio en la pila.
- La operación `PilaVacia` determina si la pila está vacía y devuelve un valor lógico verdadero en su caso.

- Cada operación de extracción de datos debe asegurarse previamente de que existe al menos un elemento de la pila. Si no existe ese elemento, la pila se encontrará en desbordamiento negativo y se producirá un error.
- `PilaLlena` determina si existe espacio en la pila para introducir al menos un elemento.
- `ContarPila` devuelve el número de elementos que existen en ese momento en la pila.
- `LiberarPila` destruye o libera la memoria de datos asignada a la pila.
- Una **cola** es una lista lineal en la que los datos se pueden insertar por un extremo denominado *Cabeza* y se elimina o borra por el otro extremo, denominado *Cola* o *Final*.
- Una cola es una estructura FIFO (*first-in, first-out*, primero en entrar, primero en salir).
- Las operaciones básicas de una cola son: `insertar, quitar, frenteCola` y `FinalCola`.
- Las pilas y las colas se pueden implementar utilizando listas enlazadas o arrays.

EJERCICIOS

18.1. Describir dos posibles aplicaciones de las pilas en programas de computadora.

18.2. ¿Cuál es la salida de este segmento de código (tipo de dato = int):

```
Pila p;
int x = 5, y = 3;
p.Insertar(8);
p.Insertar(9);
p.Insertar(y);
x = p.Quitar();
p.Insertar(18);
x = p.Quitar();
p.Insertar(22);
while (!p.PilaVacia())
{
   y = p.Quitar();
   cout << y << endl;
}
cout << x << endl;
```

18.3. Escribir los siguientes algoritmos de una pila para datos de tipo entero:

- `crear`
- `insertar`
- `quitar`
- `pila_vacía`
- `pila_llena`
- `cima_pila`
- `longitud_pila`
- `liberar_memoria_pila`

18.4. Imagine dos pilas vacías de enteros `P1` y `P2`. Dibujar un esquema que represente a las pilas después de las siguientes operaciones:

```
P1. Insertar (3);
P1. Insertar (5);
P2. Insertar (7);
P1. Insertar (9);
P2. Insertar (11);
P2. Insertar (13);
while (!P1.pilaVacia ())
{
   P1. quitar (x);
   p2. quitar (x);
}
```

18.5. Escribir una función

```
void BorrarPila (Pila& P);
```

que limpie (borre) una pila. ¿Por qué es importante que `P` se pase por referencia?

18.6. Describir dos aplicaciones de las colas en un programa informático.

18.7. Leer 10 enteros de un array, ponerlos en una pila. Imprimir la lista original y, a continuación, imprimir la pila extrayendo los elementos.

PROBLEMAS

18.1. Escribir un programa que permita determinar si una palabra o frase es un palíndromo. *Nota*: Una palabra es un palíndromo si la lectura en ambos sentidos produce el mismo resultado; ejemplo, natan.

18.2. Escribir una función `copiarPila` que copia el contenido de una pila en otra. La función debe tener dos argumentos de tipo pila, uno para la pila fuente y otro para la pila destino.

18.3. Escribir un programa que invierta el contenido de una pila.

18.4. Escribir una función que verifique si los contenidos de dos pilas son idénticos.

18.5. Escribir un programa que cree una pila a partir de una cola.

18.6. Escribir un programa que comprima una cadena de caracteres, suprimiendo todos los caracteres espacio en blanco.

18.7. Dada una cola de enteros, escribir un programa que elimine todos los enteros negativos sin cambiar los otros elementos de la cola.

18.8. Escribir un programa que invierta el contenido de una cola.

18.9. Escribir un programa que verifique los contenidos de dos colas y devuelva verdadero si son idénticas y falso en caso contrario.

18.10. Una cola de coches (carros) se describe utilizando una lista enlazada. Escribir la parte de un programa que crea una lista que describe una cola de tres coches. Por cada coche se deben almacenar número de matrícula, marca y modelo y año de matrícula.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 361).
2. Sitio web del libro, *`www.mhe.es/joyanes`* (código fuente).

18.1. *Escribir un programa que usando la clase* `Pila`*, lea datos de la entrada (–1 fin de datos), los almacene en una pila y posteriormente visualice la pila.*

18.2. *¿Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es* `int`*?*

```
Pila P;
int x = 4, y;
P.VaciaP();
P.AnadeP(x);
P.BorrarP();
P.AnadeP(32);
y = P.PrimeroP();
P.BorrarP();
P.AnadeP(y);
do
{
    cout <<" "<< P.PrimeroP()
         << endl;
    P.BorrarP();
}
while (!P.EsvaciaP());
```

18.3. *Codificar el destructor de la clase* `Pila` *implementada con punteros del Ejemplo 18.2, para que libere toda la memoria apuntada por la pila.*

18.4. *Escribir una función que no sea miembro de la clase* `Pila`*, que copie una pila en otra.*

18.5. *Escribir una función que no sea miembro de la clase* `Pila`*, que muestre el contenido de una pila que reciba como parámetro.*

18.6. *Considerar una cola de nombres representada por una array circular con 6 posiciones, y los elementos de la Cola: Mar, Sella, Centurión. Escribir los elementos de la cola y los valores de los nodos siguiente de Frente y Final según se realizan estas operaciones:*

- *Añadir Gloria y Generosa a la cola.*
- *Eliminar de la cola.*
- *Añadir Positivo.*
- *Añadir Horche a la cola.*
- *Eliminar todos los elementos de la cola.*

18.7. *Escribir una función no miembro de la clase* `Cola` *que visualice los elementos de una cola.*

18.8. *Escribir una función que reciba una cola como parámetro, e informe del número de elementos que contiene la cola.*

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 364).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

18.1. *Implementar una clase pila con un array dinámico.*

18.2. *Implementar una clase pila con las funciones miembro «pop» y «push». Ésta llena con su constructor y destructor, usando un array dinámico y tipos genéricos.*

18.3. *Escribir las funciones no miembro de la clase Pila MayorPila y MenorPila, que calculan el elemento mayor, menor de una pila de enteros.*

18.4. *Escribir la función no miembro de la clase pila, MediaPila que calcule la media de una pila de enteros.*

18.5. *Escribir una función no miembro de la clase Pila, que decida si dos pilas son iguales.*

18.6. *Escribir una función para determinar si una secuencia de caracteres de entrada es de la forma: X&Y. Donde X es una cadena de caracteres e Y es la cadena inversa. El carácter & es el separador y siempre se supone que existe. Por ejemplo, ab&ba sí es de la forma indicada, pero no lo son ab&ab ni o ab&xba.*

18.7. *Implementar la clase* `Cola` *con una lista circular simplemente enlazada.*

18.8. *Implementar una función no miembro de la clase* `Cola` *de enteros que reciba una cola como parámetro y retorne el elemento mayor y el menor de una instancia de la clase* `Cola`*.*

18.9. *Escribir una función que reciba dos objetos de la clase* `Cola` *como parámetros y decida si son iguales.*

18.10. *Escribir una función que reciba como parámetro una instancia de la clase* `Cola` *y un elemento, y elimine todos los elementos de la instancia que sean mayores que el que recibe como parámetro.*

18.11. *Escribir un programa para gestionar una cola genérica implentada con un array dinámico circular con control de excepciones.*

CAPÍTULO 19

RECURSIVIDAD

Contenido

INTRODUCCIÓN

La recursividad (recursión) es aquella propiedad que posee una función por la cual dicha función puede llamarse a sí misma. Se puede utilizar la recursividad como una alternativa a la iteración. Una solución recursiva es normalmente menos eficiente en términos de tiempo de computadora que una solución iterativa debido a las operaciones auxiliares que llevan consigo las llamadas suplementarias a las funciones; sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver. Por esta causa, la recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación.

CONCEPTOS CLAVE

- Complejidad.
- Iteracción *versus* recursión.
- Pivote.
- *Quicksort*.
- Recursividad.
- Torres de Hanoi.

19.1. LA NATURALEZA DE LA RECURSIVIDAD

Los programas examinados hasta ahora, generalmente estructurados, se componen de una serie de funciones que se llaman unas a otras de modo disciplinado. En algunos problemas es útil disponer de funciones que se llamen a sí mismas. Una *función recursiva* es una función que se llama a sí misma bien directamente o bien a través de otra función. La recursividad es un tópico importante examinado frecuentemente en cursos de programación y de introducción a las ciencias de la computación.

En este libro se dará una importancia especial a las ideas conceptuales que soportan la recursividad. En matemáticas existen numerosas funciones que tienen carácter recursivo; de igual modo numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo.

Una *función recursiva* es una función que tiene sentencias que hacen llamadas a la propia función. Hasta este momento sólo se han visto funciones que llaman a otras funciones. Así, supongamos que se dispone de dos funciones `func1` y `func2`. La organización de un programa tal y como se ha visto hasta este momento adoptaría una forma similar a ésta:

```
func1(...)
{
   ...
}
func2(...)
{
   ...
   func1(...);         // llamada a func1
   ...
}
```

Con una función recursiva, se tendría esta situación:

```
func1(...)
{
   ...
   func1(...);
   ...
}
```

Ejemplo 19.1

El factorial de un entero negativo `n`*, escrito* `n-!` *(y pronunciado n factorial), es el producto*

```
n! = n.(n - 1).(n - 2),...,1
```

en el cual

```
0! = 1
1! = 1
2! = 2 * 1
3! = 3* 2* 1
...
```

así:

```
5! = 5 * 4 * 3 * 2 * 1 = 5 * 4! = 120
         └─────────────┘
              (5-1)!
```

de modo que una definición recursiva de la función factorial *n* es:

```
n! = n*(n - 1)! = n*(n−1)*(n−2)*...*1
```

El factorial de un entero *n*, mayor o igual a 0, se puede calcular de modo *iterativo* (no recursivo), teniendo presente la definición de `n!` del modo siguiente:

```
n! = 1                si    n = 0
n! = n * (n - 1)!     si    n > 0
```

El algoritmo que resuelve la solución iterativa de un entero *n*, mayor o igual que 0, se puede calcular utilizando un bucle `for`:

```
factorial = 1;
for (int contador = n; contador >= 1; contador--)
   factorial *= contador;
```

En el caso de implementar una función se requerirá una sentencia de retorno que devuelva el valor del factorial tal como

```
return (factorial)
```

El algoritmo que resuelve la función de modo *recursivo* ha de tener presente una condición de salida. Así, en el caso del cálculo de `6!`, la definición es `6! = 6 × 5 y 5!` de acuerdo a la definición es `5 × 4!` Este proceso continúa hasta que `1! = 1 × 0!` por definición. El método de definición de una función en términos de sí misma se llama en matemáticas una definición **inductiva** y conduce naturalmente a una implementación recursiva. El caso base de `0! = 1` es esencial dado que se detiene, potencialmente, una cadena de llamadas recursivas. Este caso base o condición de salida deben fijarse en cada caso de una solución recursiva.

El algoritmo que resuelve `n!` de modo recursivo se apoya en la definición siguiente:

```
n! = 1                si      n = 0
n! = n*(n - 1)!       si      n > 0
```

en consecuencia el algoritmo mencionado que calcula el factorial será:

```
if (n == 0)
   fac = 1;
else
      fac = n * factorial (n - 1);
```

Ejemplo 19.2

Deducir la definición recursiva del producto de números naturales.

El producto $a * b$ donde a y b son enteros positivos tiene dos soluciones.

1. *Solución iterativa* $$a * b = \underbrace{a + a + a + ... + a}_{b \text{ veces}}$$

2. *Solución recursiva*

$a * b = a$	**si**	$b == 1$
$a * b = a * (b - 1) + a$	**si**	$b > 1$

Así por ejemplo, `7 × 3` será:

```
7 * 3 = 7 * 2 + 7 = 7 * 1 + 7 + 7 = 7 + 7 + 7 = 21
```

Ejemplo 19.3

Definir la naturaleza de la ***serie de Fibonacci:*** *0, 1, 1, 2, 3, 5, 8 ,13, 21, ...*

Se observa en esta serie que comienza con 0 y 1, y tiene la propiedad de que cada elemento es la suma de los dos elementos anteriores, por ejemplo:

0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
...

Entonces se puede decir que:

```
fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

y la definición recursiva será:

```
fibonacci(n) = n     si n = 0    o bien  n = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)  si n > = 2
```

Obsérvese que la definición recursiva de los números de Fibonacci es diferente de las definiciones recursivas del factorial de un número y del producto de dos números. Así por ejemplo, simplificando el nombre de la función por `fib`

```
fib(6) = fib(5) + fib(4)
```

o lo que es igual, `fib(6)` ha de aplicarse en modo recursivo dos veces, y así sucesivamente. Un algoritmo iterativo sería

```
if (n <= 1 )
   return(n);
fibinf = 0;
fibsup = 1;

for (i = 2; i <= n; i++){
  x = fibinf;
  fibinf = fibsup;
  fibsup = x + fibinf;
}                  // fin de for
return (fibsup);
```

y el algoritmo recursivo equivalente:

```
{
   if (n == 0 || n == 1)
```

```
        return n
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

19.2. FUNCIONES RECURSIVAS

Una **función recursiva** es una función que se invoca a sí misma. En **recursión directa** el código de la función `F` contiene una sentencia que invoca a `F`, mientras que en **recursión indirecta** la función `F` invoca a una función `G` que invoca a su vez a la función `H`, y así sucesivamente, hasta que se invoca de nuevo la función `F`.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia, la definición recursiva debe incluir un **componente base** (*condición de salida*) en el que *f(n)* se defina directamente (es decir, no recursivamente) para uno o más valores de *n*.

Debe existir una «forma de salir» de la secuencia de llamadas recursivas. Así, en la función *f(n) = n !* para *n* entero

$$f(n)\{ \begin{cases} 1 & n = 1 \\ n.f(n-1) & n > 1 \end{cases}$$

la condición de salida o base es $f(n) = 1$ para $n \Im 1$.

En el caso de la serie de Fibonacci

$$F_0 = 0,\ F_1 = 1, F_n = F_{n-1} + F_{n-}2 \qquad \text{para } n > 1$$

$F_0 = 0$ y $F_1 = 1$ constituyen el componente base o condiciones de salida y $F_n = F_{n-1} + F_{n-2}$ es el componente recursivo.

C++ permite escribir funciones recursivas. Una función recursiva correcta debe incluir un componente base o condición de salida.

Problema 19.1

Escribir una función recursiva que calcule el factorial de un número n y un programa que maneje dicha función.

Recordemos que

```
n! = 1                   si n = 0
n! = n.(n - 1)           si n ∀ 1
```

La función recursiva que calcula *n!*

```
int Factorial (int n)
{ // cálculo de n!
    if (n <= 1)
        return 1;
    else
        return n * Factorial(n – 1);
}
```

En el algoritmo anterior se ha considerado que el valor resultante es de tipo entero; sin embargo, observe la secuencia de valores de la función factorial

n	n!
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Como se puede ver, los valores crecen muy rápidamente, y para $n = 8$ ya sobrepasa el valor normal del mayor entero manejado en computadoras de 16 bits (32767). Por consiguiente, será preciso cambiar el tipo de dato devuelto que ha de ser `float`, `unsigned long`, etc. En consecuencia, el programa `fac.cpp` que calcula el factorial de un número puede ser:

```
// Programa fac.cpp
#include <<iostream>>
using namespace std;

#include "stdlib.h"
float factorial (int n)
{
   if (n == 0)
      return 1;
   else
   }
      float resultado = n * factorial(n – 1);
      return resultado;
   }
}

int main()
{
   cout << "Por favor introduzca un número :";
   int n;
   cin >> n;
   cout << n << " != " << factorial(n) << "\n";
   return EXIT_SUCCESS;
}
```

Una variante de este programa podría ser el cálculo del factorial correspondiente a los números naturales 0 a 10. Para ello bastaría sustituir la función `main` anterior por una función como ésta e incluyendo una llamada al archivo `#include <iomanip.h>`.

```
int main()
{
   for (int i = 0; i <= 10; i++)
      cout << setw(2) << i << "!= " << factorial(i) << endl;
   return 0;
}
```

Problema 19.2

Escribir una función de Fibonacci de modo recursivo y un programa que manipule dicha función, de modo que calcule el valor del elemento de acuerdo a la posición ocupada en la serie

```
#include <iostream>
using namespace std;

unsigned long fibonacci (unsigned long);

main()
{
   unsigned long resultado, num;

   cout << "Introduzca un entero : ";
   cin >> num;
   resultado = fibonacci (num);
   cout << "El valor de Fibonacci(" << num << ")= resultado" << endl;
   return 0 ;
}

// definición recursiva de la función de Fibonacci
unsigned long fibonacci(unsigned long n)
{
   if (n == 0 || n == 1)
      return n;
   else
      return fibonacci(n – 1) + fibonacci(n – 2);
}
```

La salida resultante de la ejecución del programa anterior:

```
Introduzca un entero : 2
El valor de Fibonacci (2) = 1

Introduzca un entero : 20
El valor de Fibonacci(20) = 832040
```

19.2.1. Funciones mutuamente recursivas

La recursividad indirecta se produce cuando una función llama a otra, que eventualmente terminará llamando de nuevo a la primera función. El programa `ALFABETO.CPP` visualiza el alfabeto utilizando recursión mutua o indirecta.

```
#include <iostream>
#include <stdio.h>
using namespace std;

void A(int c);
void B(int c);

main()
{
   A('Z');
   cout << endl;
   return 0;
}
void A(int c)
{
   if (c > 'A')
      B(c);
   putchar(c);
}

void B(int c)
{
   A(--c);
}
```

El programa principal llama a la función recursiva **`A()`** con el argumento `'Z'` (la última letra del alfabeto). La función `A` examina su parámetro *c*. Si *c* está en orden alfabético después que `'A'`, la función llama a B(), que inmediatamente llama a **`A()`**, pasándole un parámetro predecesor de *c*. Esta acción hace que **`A()`** vuelva a examinar *c*, y nuevamente una llamada a **`B()`**, hasta que *c* sea igual a `'A'`. En este momento, la recursión termina ejecutando **`putchar()`** veintiséis veces y visualizando el alfabeto, carácter a carácter.

19.2.2. Condición de terminación de la recursión

Cuando se implementa una función recursiva será preciso considerar una condición de terminación, ya que en caso contrario la función continuaría indefinidamente llamándose a sí misma y llegaría un momento en que la memoria se podría agotar. En consecuencia, sería necesario establecer en cualquier función recursiva la condición de parada que termine las llamadas recursivas y evitar indefinidamente las llamadas. Así, por ejemplo, en el caso de la función `factorial`, definida anteriormente, la condición de salida puede ser cuando el número sea 1 o 0, ya que en ambos casos el factorial es 1.

```
long factorial(long n)
{
   if (n==1) return 1;
      else return n * factorial (n-1);
}
```

19.3. RECURSIÓN FRENTE A ITERACIÓN

En los apartados anteriores se han estudiado varias funciones que se pueden implementar fácilmente o bien de modo recursivo o bien de modo interativo. En esta sección compararemos los dos enfoques y examinaremos las razones por las que el programador puede elegir un enfoque u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva y la recursión utiliza una estructura de selección*. La iteración y la recursión implican ambas repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas a funciones. La iteración y recursión implican cada una un test de terminación (*condición de salida*). La iteración termina cuando la condición del bucle no se cumple, mientras que la recursión termina cuando se reconoce un caso base o la condición de salida se alcanza.

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de recursividad y, en consecuencia, se necesita tiempo suplementario para realizar las llamadas a funciones.

Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada recursiva produce que otra copia de la función (realmente sólo las variables de función) sea creada; esto puede consumir memoria considerable. Por el contrario, la iteración se produce dentro de una función de modo que las operaciones suplementarias de las llamadas a la función y asignación de memoria adicional son omitidas.

En consecuencia, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria; es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir debe ser, normalmente, la iterativa.

> Cualquier problema que se puede resolver recursivamente, se puede resolver también iterativamente (no recursivamente). Un enfoque recursivo se elige normalmente con preferencia a un enfoque iterativo cuando el enfoque recursivo es más natural para la resolución del problema y produce un programa más fácil de comprender y depurar. Otra razón para elegir una solución recursiva es que una solución iterativa puede no ser clara ni evidente.

Consejo de programación

Se ha de evitar utilizar recursividad en situaciones de rendimiento crítico o exigencia de altas prestaciones en tiempo y memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional.

Ejemplo 19.4

La función factorial de un número ya expuesta anteriormente ofrece un ejemplo claro de comparación entre funciones definidas de modo iterativo o modo recursivo.

El factorial (*n !)*, de un número *n* era

```
0! = 1
n! = n *(n - 1)!  para    n > 0
```

Solución recursiva

```
// archivo facrecur.cpp
// Cálculo del FACTORIAL de n
```

```
int fact (int n)
// Precondición    n está definido y n >= 0
// Postcondición   ninguna
// Devuelve        n!
{
   if (n <= 0)
      return 1;
   else
      return n * fac(n - 1);
}  // fin de fac
```

Solución iterativa

```
// archivo faciter.cpp
// Cálculo de FACTORIAL de n

int fact(int n)
// Precondición    n está definida y n >= 0
// Postcondición   ninguna
// Devuelve        n!
{
   // Datos locales ...
   int factorial;

   factorial = 1;
   for (int i = 2; i <= n; i++)
      factorial *= i;
   return factorial;
}  // fin de fact
```

19.3.1. Directrices en la toma de decisión: iteración/recursión

1. Considérese una solución recursiva sólo cuando una solución iterativa *sencilla* no sea posible.
2. Utilícese una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables considerando las limitaciones del sistema.
3. Si son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales a funciones.
4. En ciertos problemas, la recursión conduce naturalmente a soluciones que son mucho más fiables de leer y comprender que su correspondiente iterativa. En estos casos los beneficios obtenidos con la claridad de la solución suelen compensar el coste extra (en tiempo y memoria) de la ejecución de un programa recursivo.

Ejemplo 19.5

Versión recursiva e iterativa de la función de Fibonacci.

La función de Fibonacci se define recursivamente de la siguiente forma:

```
Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2) si n > 1;
Fibonacci(n) = n en otro caso.
```

Para realizar la programación iterativa, hay que observar que la secuencia de números de fibonacci es: 0, 1, 1, 2, 3, 5, 8, 13 … Esta secuencia se obtiene partiendo de los números 0, 1 y a partir de ellos

cada número se obtiene sumando los dos anteriores: $a_n = a_{n-1} + a_{n-2}$. Si se definen dos variables locales `a1` y `a2`, convenientemente inicializadas a `0` y a `1` respectivamente, el siguiente valor de la serie se obtiene sumando `a1` y `a2` en otra variable local `a3`. Si ahora se hace `a1 = a2 y a2 = a3`, se tienen en estas dos variables los dos siguientes números de la serie, por lo que al iterar se van obteniendo todos los sucesivos valores.

```
long Fibonacci(int n)                                     // recursiva
{
  if (n == 0 || n == 1)
     return (n);
   else
     return(Fibonacci(n – 1) + Fibonacci(n – 2));

}


long FibonacciIterativa(int n)
{
  long a1 = 0, a2 = 1, a3, i;
  for ( i = 2; i <= n; i++)
  {
     a3 = a1 + a2;
     a1 = a2;
     a2 = a3;
  }
  return a3;
}
```

19.4. RECURSIÓN INFINITA

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de salida.

En realidad, la **recursión infinita** significa que cada llamada recursiva produce otra llamada recursiva y ésta a su vez otra llamada recursiva, y así para siempre. En la práctica dicha función se ejecutará hasta que la computadora agota la memoria disponible y se produce una terminación anormal del programa.

El flujo de control de una función recursiva requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión (*condición de salida* o *caso base*).
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

Ejemplo 19.6

Se desea calcular la suma de los primeros N enteros positivos.

La función no recursiva que realiza la tarea solicitada es:

```
int CalculoSuma (int N);
{
   int suma = 0, i;
```

```
   for (i = 1-; i <= N; i = i + 1)
        suma = suma + i;
   return suma;
}
```

La función `CalculoSuma` implementada recursivamente requiere la definición previa de la suma de los primeros `N` enteros matemáticamente en forma recursiva, tal como se muestra a continuación:

```
           ⎧ 1                    si   N = 1
suma(N) =  ⎨
           ⎩ N + suma(N-1)        en caso contrario
```

La definición anterior significa que si `N` es `1`, entonces la función `suma(N)` toma el valor 1. En caso contrario, significa que la función `suma(N)` toma el valor resultante de la suma de `N` y el resultado de `suma(N-1)`. Por ejemplo, la función `suma(5)` se evalúa tal como se muestra en la Figura 19.1.

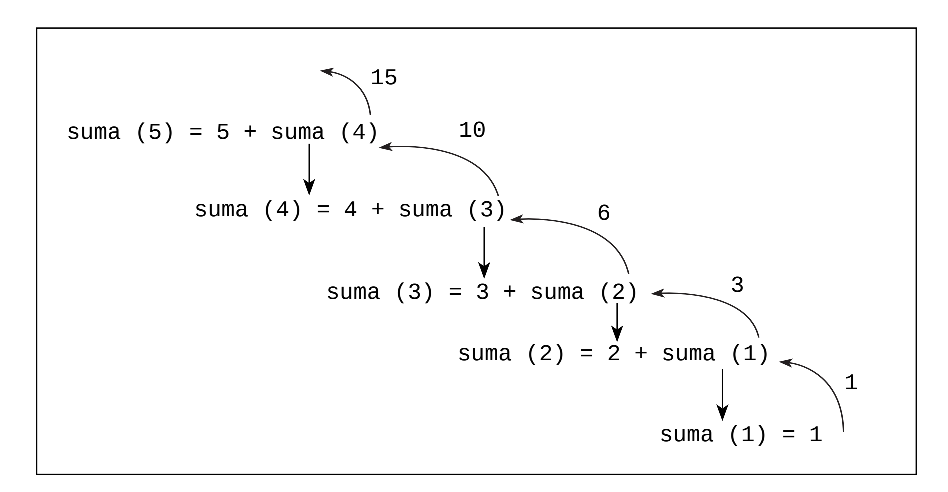


Figura 19.1. Secuencia de llamadas recursivas que evalúan la función Suma(N) (en el ejemplo Suma(5)).

El código fuente de la función recursiva `suma` es:

```
int suma(int n)
{
   if (n == 1)                  test para parar o continuar (condición de salida)
       return 1;                caso final - se detiene la recursión
   else
   return n + suma(n - 1);      caso recursivo - la recursión continúa con una llamada recursiva
}
```

Cuando se realizan llamadas recursivas se han de pasar argumentos diferentes de los parámetros de entrada; así, en el ejemplo de la función `suma`, el argumento que se pasa en la función recursiva es *n – 1* y el parámetro es *n*.

La Figura 19.2 muestra el flujo de control de la función `suma` de modo recursivo.

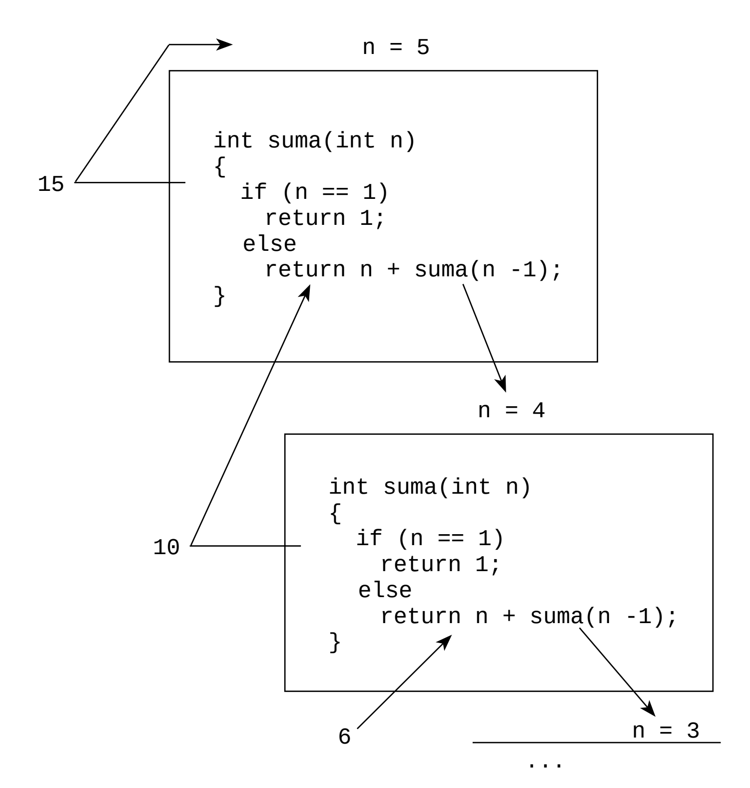


Figura 19.2. Flujo de control de la función `suma` recursiva.

Problema 19.3

Deducir una función `mcd()` *que calcule el mayor denominador común o máximo común divisor de dos números enteros* `b1` *y* `b2` *(el* **mcd**, *máximo común divisor, es el entero mayor que divide a ambos números) y un programa que la manipule.*

Consideremos el cálculo del *máximo común divisor* (**mcd**) de dos enteros. El *mcd* de los enteros *a* y *b* se define como el entero mayor que divide a ambos números. El mcd no está definido si *a* y *b* son cero. Los valores negativos de *a* y *b* se sustituyen por sus valores absolutos. El algoritmo recursivo que calcula *mcd(a, b)* se describe con el siguiente pseudocódigo:

1. Si *b* es cero, la solución es *a*.
2. Si *b* no es cero, la solución es *mcd(b, a mod b).*

La función recursiva es:

```
int mcd(int a, int b)
{
   if (b == 0)
     return (a);
   else
     return (mcd(b, a%b));
}
```

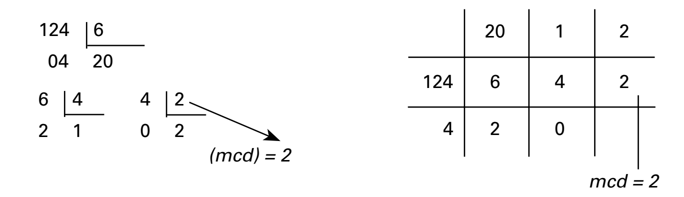
Los pasos del método para el caso de los números 2970 y 1265 serían los siguientes:

2970 |1265   1265 |440   440 |385
0440   2   385   2   55   1

385 |(55)
00   7   *mcd* = 55

mcd(2970, 1265) = 55

Supongamos otro ejemplo con los números 6 y 124; el procedimiento clásico de obtención del **mcd** es la obtención de divisiones sucesivas entre ambos números 124 entre 6, si el resto no es 0, se divide el número menor por el resto, y así sucesivamente hasta que el resto sea 0, en cuyo caso el **mcd** es el último valor del divisor.



En el caso de 124 y 6, el **mcd** es 2. En consecuencia, la condición de salida es que el resto sea cero. El código fuente de la función es:

```
int mcd(int b1, int b2)
{
   if (b2!= 0)        // condición de salida
     return mcd (b2, b1%b2);
   return b1;
}
```

El algoritmo del **mcd** entre dos números *m* y *n* es:

- `mcd(m, n)` es `n` si `n <= m` y `n` divide a `m`
- `mcd(m, n)` es `mcd(n, m)` si `m` < `n`
- `mcd(m, n)` es `mcd(n, resto de m dividido por n` en caso contrario.

Los pasos anteriores significan: el **mcd** es `n` si `n` es el número más pequeño y `n` divide a `m`. Si `m` es el número más pequeño, entonces la determinación del *mcd* se debe ejecutar con los argumentos transpuestos. Por último, si `n` no divide a `m`, el resto se obtiene encontrando el `mcd` de `n` y el resto de `m` divido por `n`.

Un programa que gestiona la función `mcd` es `mcd.cpp`.

```
#indice <iostream>
using namespace std;

int mcd(int, int);

void main()
{
   // datos locales
   int m, n;
```

```
    cout << "Introduzca dos enteros positivos:"
    cin >> m >> n;
    cout << endl;
    cout << "El máximo común divisor es: " << mcd (m, n) << endl;
}

// Función recursiva mcd
int mcd(int m, int n)
// m y n han de ser mayores > 0
// devuelve el máximo común divisor de m y n
{
      if (n <= m && m % n == 0)
         return n;
      else if (m < n)
         return mcd(n, m);
      else
         return mcd(n, m % n);
}      // final de mcd
```

Al ejecutarse el programa se produce la siguiente salida

```
Introduzca dos enteros positivos: 6   40
El máximo común divisor es: 2
```

Ejemplo 19.7

Función recursiva que calcula la suma de los cuadrados de los N primeros números positivos.

La función `sumacuadrados` implementada recursivamente, requiere la definición previa de la suma de los cuadrados primeros `N` enteros matemáticamente en forma recursiva tal como se muestra a continuación:

$$\text{suma}(N) = \begin{cases} 1 & \text{si} \quad N = 1 \\ N^2 + \text{suma}(N-1) & \textit{en caso contrario} \end{cases}$$

```
int sumacuadrados(int n)
{
  if (n == 1)        //test para parar o continuar (condición de salida)
    return 1;                     //caso final se detiene la recursión
  else
  return n*n + sumacuadrados (n – 1);      //caso recursivo la recursión
                                       //continúa con llamada recursiva
}
```

Esta recursión para siempre que se llame a la función con un valor de `n` positivo, pero se produce una recursividad infinita en caso de que se llame con un valor de `n` negativo o nulo.

Ejemplo 19.8

Función recursiva que calcula el producto de dos números naturales, usando sumas.

El producto de dos números naturales `a` y `b` se puede definir recursivamente de la siguiente forma: `Producto(a, b)= 0 si b = 0; Producto(a, b) = a + Producto(a, b – 1) si b > 0`.

Esta función tendrá una llamada recursiva infinita si es llamada con el valor del segundo parámetro negativo.

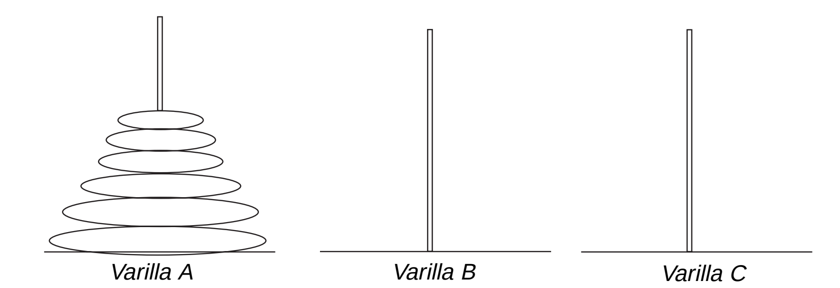
```
int Producto(int a, int b)
 {
   if (b == 0)
      return 0;
   else
      return a + Producto(a, b – 1);
 }
```

19.5. RESOLUCIÓN DE PROBLEMAS CON RECURSIÓN

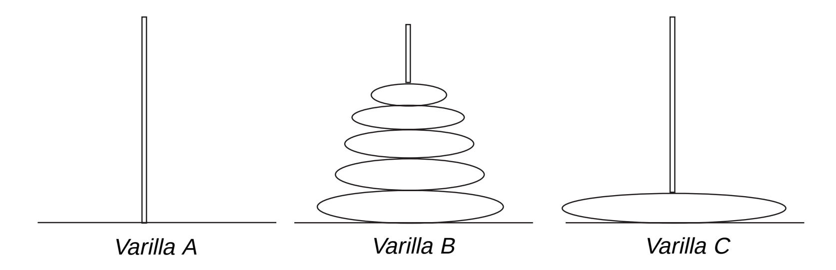
Muchos problemas de computadora tienen una formulación simple y elegante que se traduce directamente a código recursivo. En esta sección se describen una serie de ejemplos que incluyen problemas clásicos resueltos mediante recursividad. Entre ellos se destacan problemas matemáticos, las Torres de Hanoi, método de búsqueda binaria, ordenación rápida, árboles de expresión, etc. Explicaremos con detalle algunos de ellos.

19.5.1. Torres de Hanoi

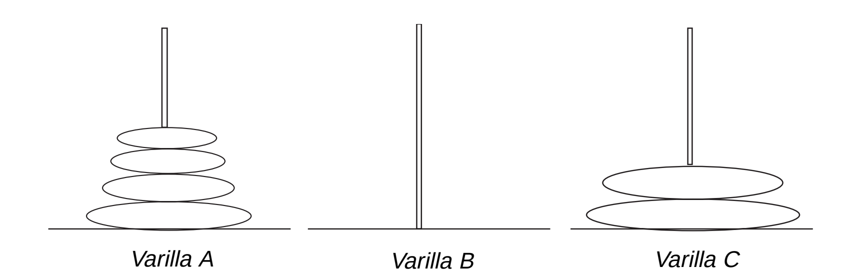
Este juego (un algoritmo clásico) tiene sus orígenes en la cultura oriental y en una leyenda sobre el templo de Brahma cuya estructura simulaba una plataforma metálica con tres varillas y discos en su interior. El problema en cuestión suponía la existencia de tres varillas o postes (A, B y C) en los que se alojaban discos (*n* discos) que se podían trasladar de una varilla a otra libremente pero con una condición: cada disco era ligeramente inferior en diámetro al que estaba justo debajo de él.



Ilustremos el problema con tres varillas que contengan seis discos en una varilla A, y se desea trasladar a la varilla C conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él. Así, por ejemplo, se pueden cambiar cinco discos de golpe de la varilla A a la varilla B, y el disco más grande a la varilla C.



Ahora el problema se centra en pasar los cinco discos de la varilla B a la varilla C y se utiliza un método similar al anterior, pasar los cuatro discos superiores de la varilla B a la varilla A y, a continuación, se pasa el disco de mayor tamaño de la varilla B a la varilla C y así sucesivamente. El proceso continúa de modo recursivo hasta que finalmente se queda un disco en la varilla B que es la condición de parada.



Este problema es claramente recursivo y se puede describir así: *el juego consta de tres varillas (alambres) denominadas varilla inicial, varilla central y varilla final.*

En la pila inicial se sitúan *N* discos que se apilan en orden creciente de tamaño con el mayor en la parte inferior. El objetivo del juego o puzzle es mover los *N* discos desde la varilla inicial a la varilla final. Los discos se mueven uno a uno con la condición de que un disco mayor nunca puede ser situado encima de un disco más pequeño.

Diseño del algoritmo

El ejemplo anterior de seis discos se puede generalizar a un algoritmo recursivo con *n* discos y tres varillas. La función de Hanoi declara las varillas o postes como objetos char. En la lista de parámetros, el orden de las variables o varillas es:

```
varinicial          varcentral          varfinal
```

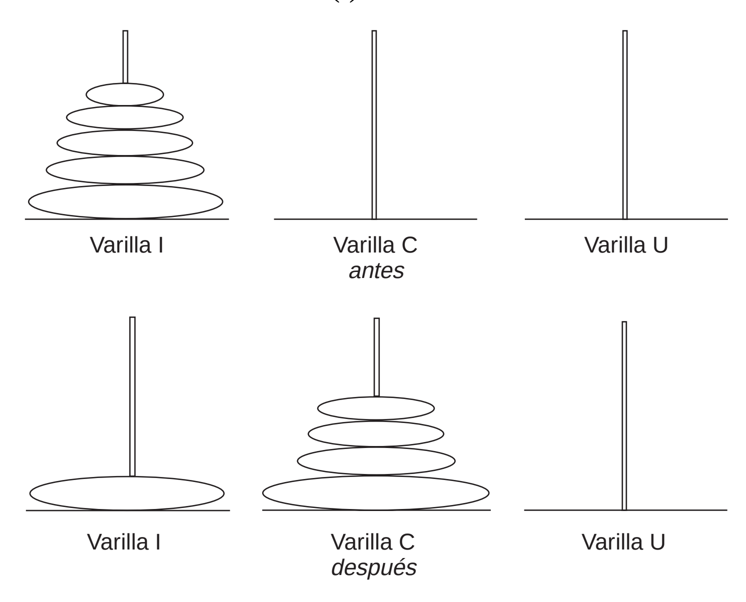
lo que implica que se están moviendo discos desde la varilla inicial a la final utilizando la varilla central como auxiliar para almacenar los discos. Si *n = 1* se tiene la condición de parada, ya que se puede manejar moviendo el único disco desde la varilla inicial a la varilla final. El algoritmo sería el siguiente:

```
1. Si n es 1
     1.1 Mover el disco 1 de varinicial a varfinal.

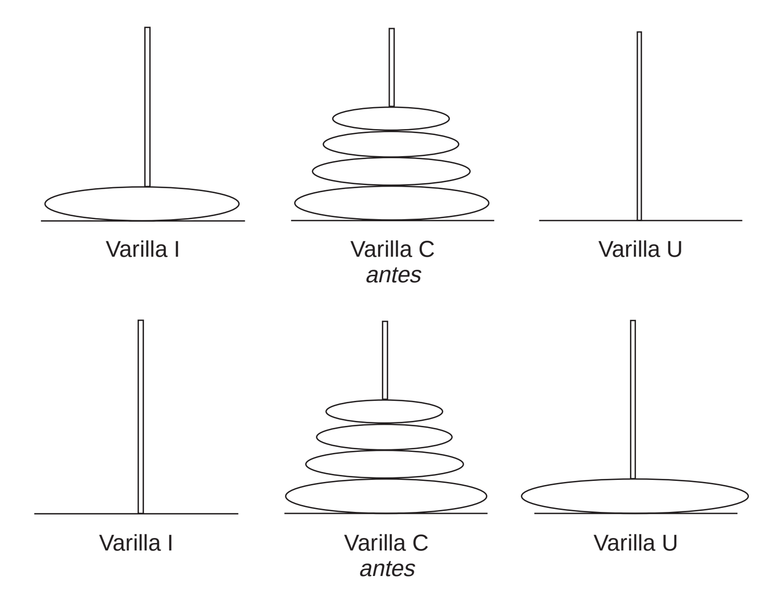
2. Sino
     1.2 Mover n - 1 discos desde varinicial hasta la varilla auxiliar uti-
         lizando varfinal.
     1.3 Mover el disco n desde varilla inicial varinicial a varfinal.
     1.4 Mover n - 1 discos desde la varilla auxiliar o central a varfinal
         utilizando la varilla inicial.
```

Es decir, si n es 1, se alcanza la condición de salida o terminación del algoritmo. Si n es mayor que 1, las etapas recursivas 1.2, 1.3 y 1.4 son tres subproblemas más pequeños, uno de los cuales es la condición de salida.

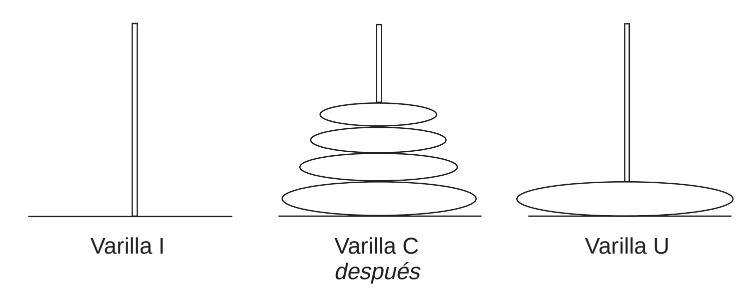
Etapa 1: Mover *n-1* discos desde varilla inicial (I).

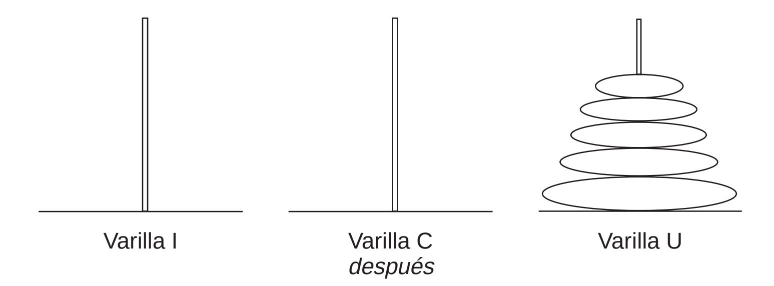


Etapa 2: Mover un disco desde I a U.



Etapa 3: Mover *n-1* discos desde C a U.





La primera etapa en el algoritmo mueve *n-1* discos desde la varilla inicial a la varilla central utilizando la varilla final. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es *varinicial*, *varfinal* y *varcentral*.

```
// utilizar varfinal como almacenamiento auxiliar
Hanoi(n - 1, varinicial, varfinal, varcentral);
```

La segunda etapa mueve simplemente el disco mayor desde el punto de la varilla inicial a la varilla final:

```
cout << "mover" << varinicial << "a" << varfinal << endl;
```

La tercera etapa del algoritmo mueve *n-1* discos desde la varilla central a la varilla final utilizando `varinicial` para almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es: `varcentral, varinicial` y `varfinal`.

```
// utilizar varinicial como almacenamiento auxiliar
Hanoi( n- 1, varcentral, varinicial, varfinal);
```

Implementación de las torres de Hanoi

La implementación del algoritmo se apoya en los nombres de las tres varillas o alambres —`"inicial"`, `"central"` y `"final"`— que se pasan como parámetros a la función. El programa comienza solicitando al usuario que introduzca el número de discos *N*. Se llama a la función recursiva `Hanoi` para obtener un listado de los movimientos que transferirá los *N* discos desde la varilla `"inicial"` a la varilla `"final"`. El algoritmo requiere 2N – 1 movimientos. Para el caso de 10 discos, el juego requerirá 1.023 movimientos. En el caso de prueba para `N = 3`, el número de movimientos es $2^3 - 1 = 7$.

```
// archivo Torres.cpp
// función recursiva Torres de Hanoi

void Hanoi(char varinicial, char varfinal, char varcentral, int n)
{
   if ( n == 1)
      cout << "Mover disco 1 de varilla" << varinicial
           << "a varilla" << varfinal << endl;
   else
   {
      Hanoi(varinicial, varcentral, varfinal, n - 1);
      cout << "Mover disco" << n << "desde varilla" <<
           <<  varinicial <<" a varilla" << varfinal << endl;
      Hanoi(varcentral, varfinal, varinicial, n - 1);
   }
}
```

Una ejecución de la función para el caso de mover tres discos desde las varillas `A` a `C` tomando la varilla `B` como varilla central o auxiliar, se puede seguir con la siguiente sentencia:

```
Hanoi ('A', 'C', 'B', 3);
```

que resuelve el problema de tres discos desde `A` a `C`. La salida generada sería:

```
Mover disco 1 de varilla A a varilla C
Mover disco 2 de varilla A a varilla B
Mover disco 1 de varilla C a varilla B
Mover disco 3 de varilla A a varilla C
Mover disco 1 de varilla B a varilla A
Mover disco 2 de varilla B a varilla C
Mover disco 1 de varilla A a varilla C
```

Consideraciones de eficiencia en la torres de Hanoi

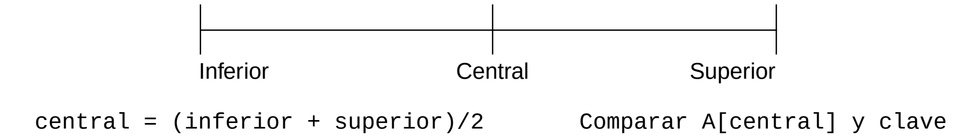
Es de destacar que la función `Hanoi` resolverá el problema de las torres de Hanoi para cualquier número de discos. El problema de tres discos se resuelve en un total de 7 ($2^3 - 1$) llamadas a la función `Hanoi` mediante siete movimientos de disco. El problema de cinco discos se resuelve con 31 ($2^5 - 1$) llamadas y 31 movimientos. En general, como ya se ha expresado anteriormente, el número de movimientos, requeridos para resolver el problema de *n* discos, es $2^n - 1$. Cada llamada a la función requiere la asignación e inicialización de un área local de datos en la memoria, por lo que el tiempo de computadora se incrementa exponencialmente con el tamaño del problema. Por estas razones, la ejecución del programa con un valor de *n* mayor que 10 requiere gran cantidad de prudencia para evitar desbordamientos de memoria y ralentización de tiempo.

19.5.2. Búsqueda binaria recursiva

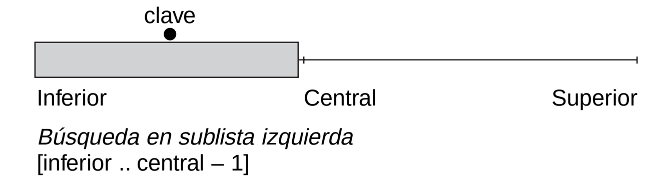
Recordemos que la búsqueda binaria era aquel método de búsqueda de una clave especificada dentro de una lista o array ordenado de *n* elementos que realizaba una exploración de la lista hasta que se encontraba o no la coincidencia con la clave especificada. El algoritmo de búsqueda binaria se puede describir recursivamente.

Supóngase que se tiene una lista ordenada `A` con un límite inferior y un límite superior. Dada una clave (valor buscado) se comienza la búsqueda en la posición central de la lista (índice central).

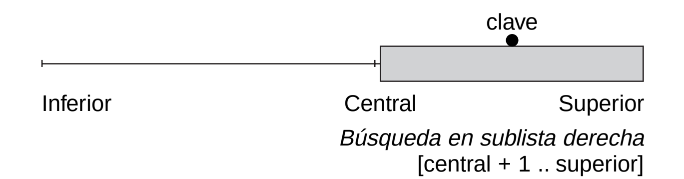
Si se produce coincidencia (se encuentra la clave), se tiene la condición de terminación que permite detener la búsqueda y devolver el índice central. Si no se produce la coincidencia (no se encuentra la clave), dado que la lista está ordenada, se centra la búsqueda en la «sublista inferior» (a la izquierda de la posición central) o en la «sublista derecha» (a la derecha de la posición central).



1. Si `clave < A[central]`, el valor buscado sólo puede estar en la mitad izquierda de la lista con elementos en el rango inferior a `central - 1`.



2. Si `clave > A[central]`, el valor buscado sólo puede entrar en la mitad derecha de la lista con elementos en el rango de índices, `central + 1` a `Superior`.



3. El proceso recursivo continúa la búsqueda en sublistas más y más pequeñas. La búsqueda termina, o con éxito (*aparece la clave buscada*) o sin éxito (*no aparece la clave buscada*), situación que ocurrirá cuando el límite superior de la lista sea más pequeño que el límite superior. La condición `Inferior > Superior` será la condición de salida o terminación y el algoritmo devuelve el índice –1.

En notación matemática y algorítmica se podría representar la búsqueda binaria de la siguiente forma:

```
BusquedaBR(inferior, superior, clave) // BR, binaria recursiva

      return  NOT_FOUND // no encontrada
                if inferior > superior
      return central
                if elemento[central] == clave
=     return BusquedaBR(central + 1, superior, clave)
s               if elemento[central] < clave
      return BusquedaBR(inferior, central - 1, clave
                if elemento[central] > clave
```

en donde `central` es el punto central entre `inferior` y `superior`. Expresado como una función C++ podría ser:

```
int BusquedaBR(int inferior, int superior, int clave)
{
   int i;
   if (inferior > superior)        // no encontrado
      return NOT_FOUND
   else {
      i = (inferior + superior)/2;
      if (elemento[i] == clave)    // encontrado
          return i;
      else if (elemento[i] < clave)
          return BusquedaBR (i+1, superior, clave);
      else
          return BusquedaBR(inferior, i-1, clave);
   }
}
```

20.6. ORDENACIÓN RÁPIDA (`quicksort`)

El algoritmo conocido como *quicksort* (ordenación rápida) recibe el nombre de su autor, Tony Hoare. La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar. El método es, posiblemente, el más pequeño de código, más rápido, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación.

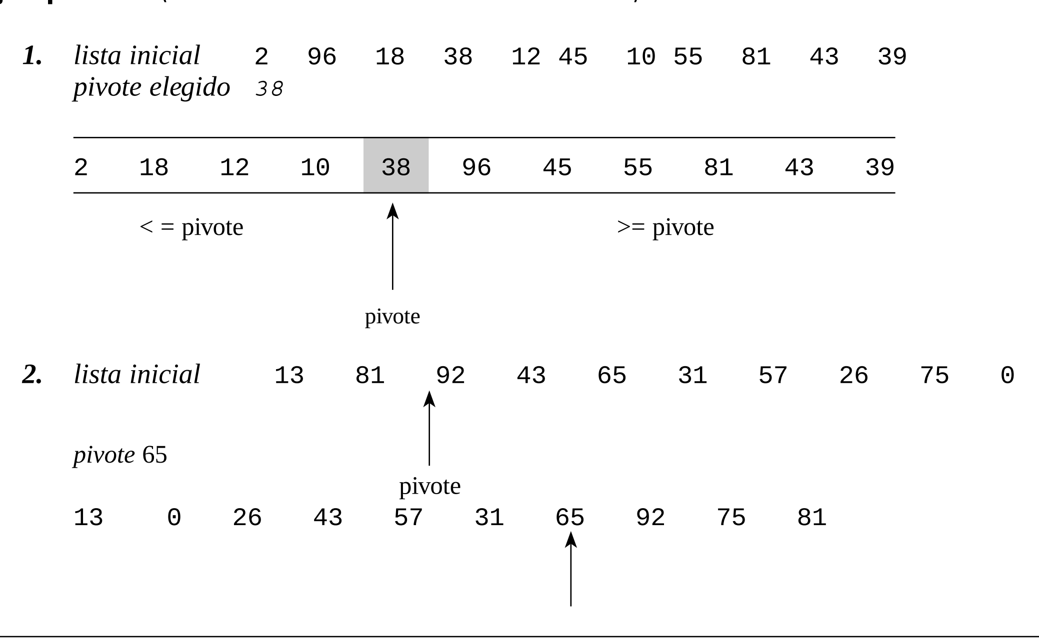
El método se basa en dividir los *n* elementos de la lista a ordenar en tres partes o particiones: una partición *izquierda*, una partición *central*, que sólo contiene un elemento denominado *pivote* o elemento de partición, y una partición *derecha*. La partición o división se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha). Las dos sublistas se ordenan entonces independientemente.

La lista se divide en particiones (sublistas) eligiendo uno de los elementos de la lista y se utiliza como *pivote* o *elemento de partición*. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede elegir cualquier elemento de la lista como pivote, por ejemplo el primer elemento de la lista. Si la lista tiene algún orden parcial, que se conoce, se puede tomar otra decisión para el pivote. Idealmente, el pivote se debe elegir de modo que se divida la lista exactamente por la mitad, de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones «pobres» de pivotes.

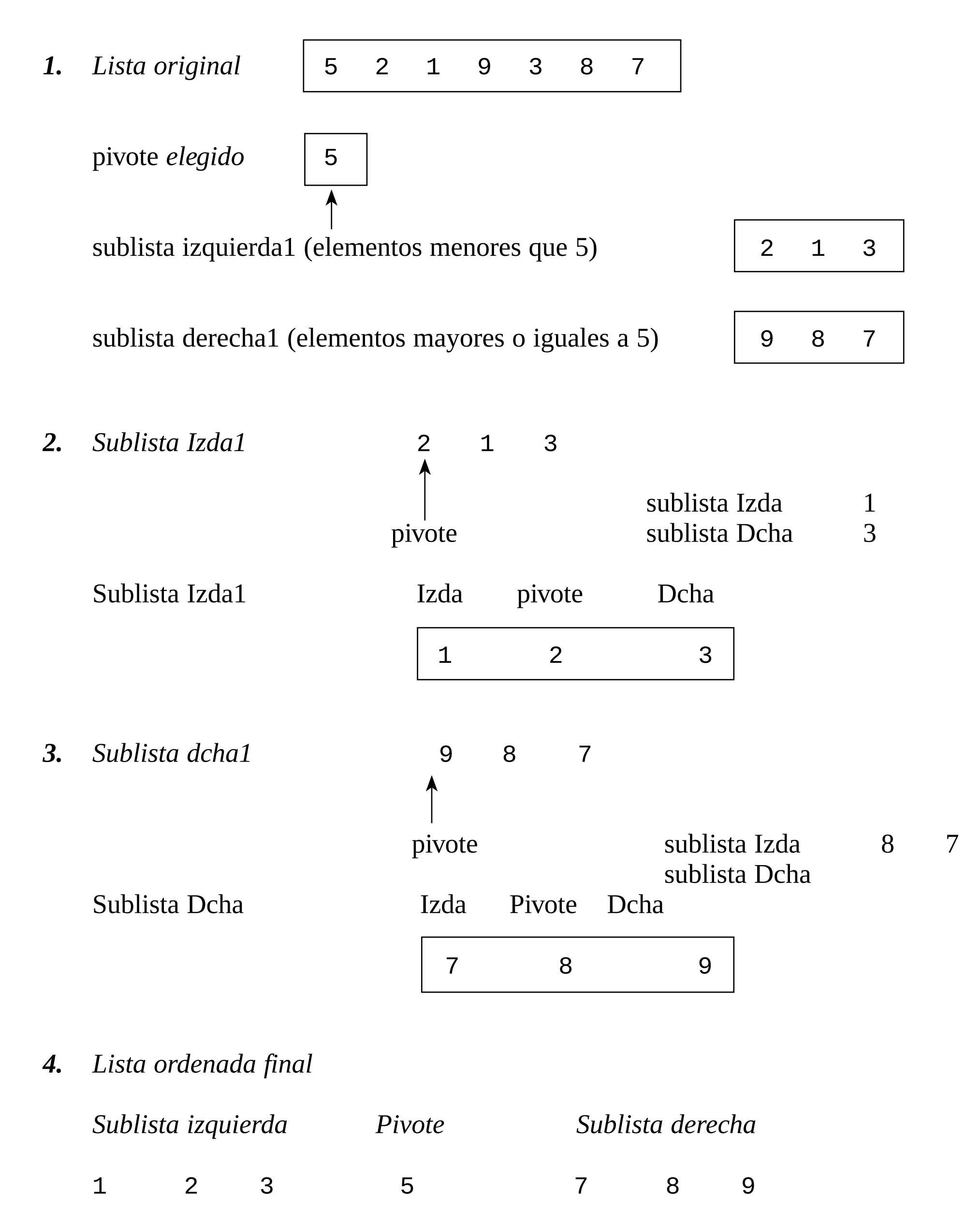
Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el pivote y la otra en la que todos los elementos (claves) son mayores o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista. La primera etapa de *quicksort* es la división o «particionado» recursivo de la lista hasta que todas las sublistas constan de solo un elemento.



Ejemplo 19.9 (Pivote: elemento interior de la lista)



Ejemplo 19.8 (Pivote: primer elemento de la lista)



El algoritmo *quicksort* requiere una estrategia de partición y la selección idónea del pivote. Las etapas fundamentales del algoritmo dependen del pivote elegido aunque la estrategia de partición suele ser similar. Supongamos que todos los elementos de la lista son distintos, aunque será preciso tener en cuenta los casos en que existan elementos idénticos. La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar el sistema para situar en la sublista izquierda todos los elementos menores o iguales que el pivote y en la sublista derecha todos los elementos mayores que el pivote, Se intercambia el pivote con el último elemento. El resultado de este intercambio será una nueva lista.

Ejemplo 19.10

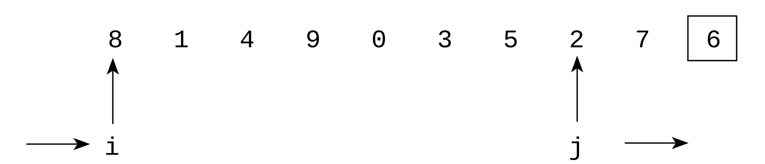
Lista original:	`8 1 4 9 6 3 5 2 7 0`
pivote (elemento central)	`6`
intercambio de 6 con elemento extremo	`0`

La etapa 2 requiere mover todos los elementos menores al pivote a la parte izquierda del array y los elementos mayores a la parte derecha.

```
8    1    4    9    0    3    5    2    7    6
```

Para ello se recorre la lista de izquierda a derecha utilizando un contador `i` que se inicializa en la posición más baja (`Inferior`) buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un contador `j` inicializado en la posición más alta: `Superior-1`.

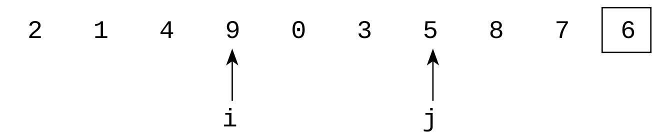
El contador `i` se detiene en el elemento 8 (mayor que el pivote) y el contador `j` se detiene en el elemento 2 (menor que el pivote).



Ahora se intercambian 8 y 2 para que estos dos elementos se sitúen correctamente en cada sublista.

```
2    1    4    9    0    3    5    8    7    [6]
```

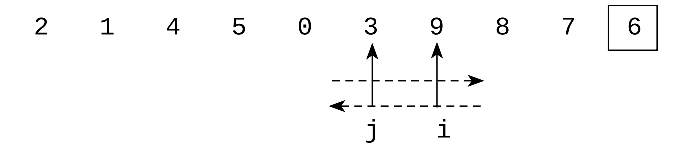
A medida que el algoritmo continúa, `i` se detiene en el elemento mayor, 9, y `j` se detiene en el elemento menor, 5,



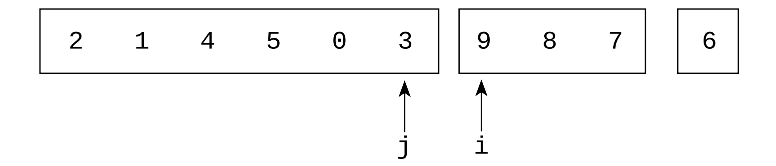
Se intercambian los elementos mientras que `i` y `j` no se cruzan. En caso contrario se detiene este bucle. En el caso anterior se intercambian 9 y 5.

```
2    1    4    5    0    3    9    8    7    [6]
```

Continúa la exploración y ahora el contador `i` se detiene en el elemento mayor 9 y el contador `j` se detiene en el elemento menor 3.

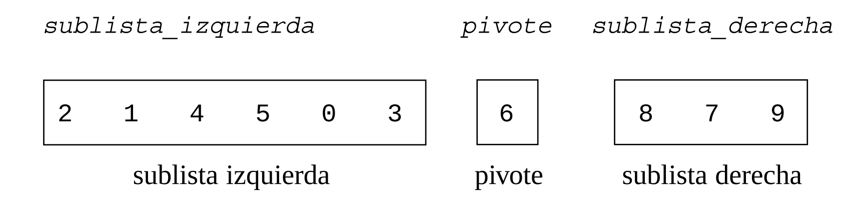


En esta posición los contadores `i` y `j` han cruzado posiciones en el array y en este caso se detiene la búsqueda y no se realiza ningún intercambio ya que el elemento al que accede el contador `j` estará ya correctamente situado. Las dos sublistas ya han sido creadas (la lista original se ha dividido en dos particiones).



Ahora ya lo único que se necesita es intercambiar el elemento que está en la posición `i` con el elemento de la última posición el pivote.

La etapa tercera requiere intercambiar el elemento de la posición `i` con el pivote, de modo que se tendrá la secuencia prevista inicialmente:



19.6.1. Algoritmo *quicksort* en C++

El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el pivote. Aunque la posición del pivote, en principio, puede ser cualquiera, una de las decisiones más ponderadas es aquella que considera el pivote como el elemento central o próximo al central de la lista. La Figura 19.3 muestra las operaciones del algoritmo para ordenar la lista de elementos enteros `L`.

```
// algoritmo quicksort
// ordenar a[0:n-1]

Seleccionar un elemento de a[0:n-1] como elemento central
   (este elemento es el pivote).
Dividir los elementos restantes en particiones izquierda y
   derecha, de modo que ningún elemento de la izquierda tenga
   una clave (valor) mayor que el pivote y que ningún elemento
    a la derecha tiene una clave más pequeña que la del pivote.
Ordenar la partición izquierda utilizando quicksort
   recursivamente.
Ordenar la partición derecha utilizando quicksort
   recursivamente.
La solución es partición izquierda seguida por el pivote y, a
   continuación, partición derecha.
```

El programa `quicksort` refleja el algoritmo *quicksort* citado anteriormente. Utilizaremos una plantilla `template` de funciones.

```
template <class T>
void quicksort(T *a, int n)
{
   // Ordenar a[0:n-1] utilizando ordenación rápida
   // Requiere que a[n] debe tener la clave inferior
   quicksort(a, 0, n-1);
}
```

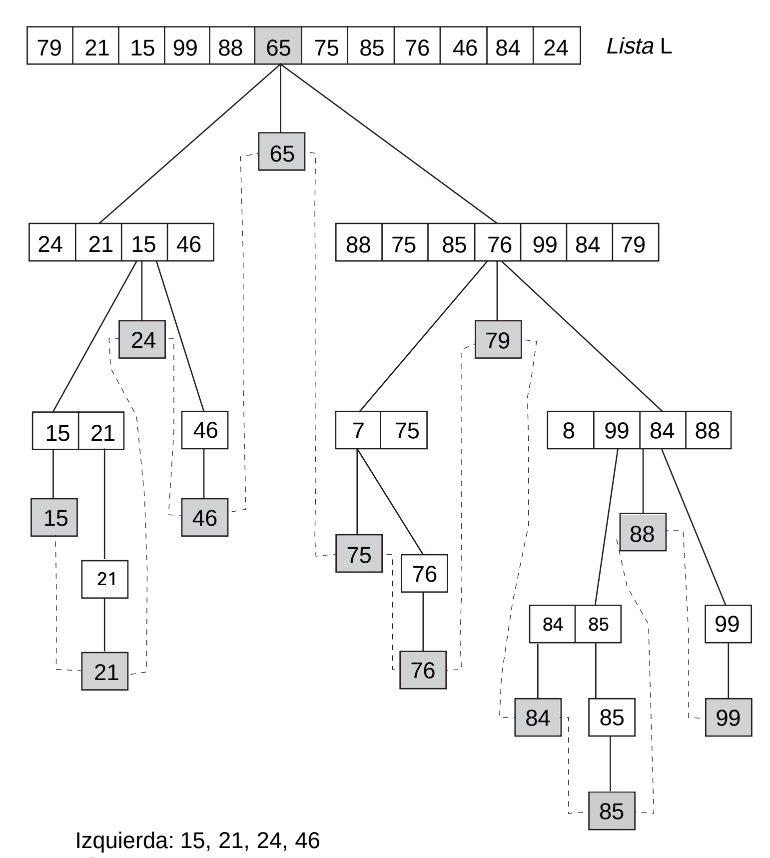


Figura 19.3. Algoritmo de ordenación *quicksort.*

```
template <class T>
void quicksort(T a[], int i, int d)
{
   // Ordenar a[i:d], a[d+1] tiene valor mayor.
   if (i >= d) return;
   int m = i;                 // cursor izquierda a derecha
   int n = d+1 ;              // cursor derecha a izquierda

   T pivote = a[i];

   // intercambiar elementos >= pivote en lado izquierdo
   // con elementos           <= pivote en lado derecho
   while (true) {
      do { // encontrar >= elemento en el lado izquierdo
          m = m+1;
      } while (a[m] < pivote);
      do { // encontrar <= elemento en el lado derecho
          n = n-1;
      } while (a[n] > pivote);
```

```
        if (m >= n) break ;    // intercambio, par no encontrado
        Intercambio (a[m], a[n]);
    }

  // situar pivote
   a[i] = a[n];
   a[n] = pivote ;

   quicksort(a, i, n-1);      // ordenar segmento izquierdo
   quicksort(a, n+1, d);      // ordenar segmento derecho
}
```

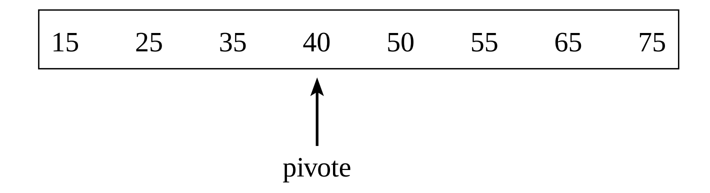
19.6.2. Análisis del algoritmo *quicksort*

El análisis general de la eficiencia de *quicksort* es difícil. La mejor forma de ilustrar y calcular la complejidad del algoritmo es considerar el número de comparaciones realizadas teniendo en cuenta circunstancias ideales. Supongamos que n (número de elementos de la lista) es una potencia de 2, $n = 2^k$ ($k = log_2 n$). Además, supongamos que el pivote es el elemento central de cada lista, de modo que *quicksort* divide la sublista en dos aproximadamente iguales.

En la primera exploración o recorrido hay $n-1$ comparaciones. El resultado de la etapa crea dos sublistas aproximadamente de tamaño $n/2$. En la siguiente fase, el proceso de cada sublista requiere aproximadamente $n/2$ comparaciones. Las comparaciones totales de esta fase son $2(n/2) = n$. La siguiente fase procesa cuatro sublistas que requieren un total de $4(n/4)$ comparaciones, etc. Eventualmente, el proceso de división termina después de k pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente:

$$n + 2(n/2) + 4(n/4) + ... + n(n/n) = n + n + ... + n$$
$$= n * k = n * log_2 n$$

Para una lista normal la complejidad de *quicksort* es $0(nlog_2 n)$. El caso ideal que se ha examinado se realiza realmente cuando la lista (el array) está ordenado en orden ascendente. En este caso el pivote es precisamente el centro de cada sublista.



Si el array está en orden ascendente, el primer recorrido encuentra el pivote en el centro de la lista e intercambia cada elemento en las sublistas inferiores y superiores. La lista resultante está casi ordenada y el algoritmo tiene la complejidad $0(nlog_2 n)$.

El escenario del caso peor de *quicksort* ocurre cuando el pivote cae consistentemente en una sublista de un elemento y deja el resto de los elementos en la segunda sublista. Esto sucede cuando el pivote es siempre el elemento más pequeño de su sublista. En el recorrido inicial, hay n comparaciones y la sublista grande contiene $n-1$ elementos. En el siguiente recorrido, la sublista mayor requiere $n-1$ comparaciones y produce una sublista de $n-2$ elementos, etc. El número total de comparaciones es

$$n + n-1 + n-2 +...+ = n(n+1)/2-1$$

La complejidad es $0(n^2)$. En general el algoritmo de ordenación tiene como complejidad media $0(nlog_2n)$ siendo posiblemente el algoritmo más rápido. La Tabla 19.1 muestra las complejidades de los algoritmos empleados en los métodos explicados en el libro.

Tabla 19.1. Comparación complejidad métodos de ordenación

Método	Complejidad
Burbuja	n^2
Inserción	n^2
Selección	n^2
Montículo	$nlog_2n$
Fusión	$nlog_2n$
Shell	$nlog_2n$
Quicksort	$nlog_2n$

En conclusión, se suele recomendar que para listas pequeñas, los métodos más eficientes son: inserción y selección, y para listas grandes, el *quicksort*. El algoritmo de Shell suele variar mucho su eficiencia en función de la variación del número de elementos, por lo que es más difícil que en los otros métodos proporcionar un consejo eficiente. Los métodos de fusión y por montículo suelen ser muy eficientes para listas muy grandes.

RESUMEN

Recursividad o *recursión* es la capacidad de una función de llamarse así misma dentro del propio cuerpo de la función. La recursividad es una alternativa a la iteración mediante el uso de bucles. Las funciones recursivas iteran haciendo llamadas recursivas. Los aspectos más importantes a tener en cuenta en el diseño y construcción de funciones recursivas son los siguientes:

- Un algoritmo recursivo de una definición de función normalmente contiene dos tipos de casos: uno o más casos que incluyen al menos una llamada recursiva y uno o más casos de terminación o parada en los que el problema se soluciona sin ninguna llamada recursiva. Dicho de otro modo, una función recursiva debe tener dos partes: una parte de terminación que termina la recursión y una llamada recursiva con sus propios parámetros y variables locales.
- Muchos problemas tienen naturaleza recursiva y deben ser resueltos con funciones recursivas ya que resultará difícil su solución utilizando bucles. De igual modo aquellos problemas que no entrañen una solución recursiva se deberán seguir resolviendo mediante soluciones iterativas.
- Las funciones llamadas recursivamente utilizan memoria; existe un límite en el número de llamadas recursivas o funciones llamadas recursivamente. Este límite depende de la cantidad de memoria de la computadora.
- Cuando se escribe una función recursiva se debe comprobar siempre que la función tiene condición de terminación; es decir, que no se producirá recursividad infinita. Ésta es la condición de error más usual durante el aprendizaje de la recursividad.
- Para asegurarse de que el diseño de una función recursiva es correcto se deben cumplir las siguientes tres condiciones:

1. No existe recursión infinita. (Una llamada recursiva puede conducir a otra llamada recursiva y ésta puede conducir a otra, y así sucesivamente, pero cada cadena de llamadas recursivas debe alcanzar, eventualmente, una condición de terminación.)
2. Cada condición de terminación devuelve el valor correcto de ese caso.
3. En los casos que implican recursión: *si* todas las llamadas recursivas devuelven el valor correcto, *entonces* el valor final devuelto por la función es el valor correcto.

EJERCICIOS

19.1. Convierta la siguiente función iterativa en una recursiva. La función calcula un valor aproximado de *e*, la base de los logaritmos naturales, sumando las series

1 + 1/1 ! + 1/2 ! + ... + 1/n !

hasta que los términos adicionales no afecten a la aproximación

```
float loge()
{
    // Datos locales
    float enl, delta, fact;
    int n;

    enl = 1.0;
    n = 1;
    fact = 1.0;
    delta = 1.0;
    do
    {
       enl + delta;
       n++;
       fact * = n:
       delta = 1.0 / fact;
    } while (enl != enl + delta);
    return enl;
}          // fin de loge
```

19.2. Explique porqué la siguiente función puede producir un valor incorrecto cuando se ejecute:

```
long factorial (long n)
{
   if (n == 0 || n == 1)
     return 1;
   else
     return n * factorial (--n);
}
```

19.3. ¿Cuál es la secuencia numérica generada por la función recursiva `f` en el listado siguiente?

```
long f(int n)
{
   if (n == 0|| n == 1)
     return 1;
   else
     return 3 * f(n - 2) + 2 * f(n
     - 1);
}
```

19.4. Escribir una función recursiva `int longcadena(char S[]);` que calcula la longitud de una cadena.

Sugerencia: Condición de salida `S[] == 0` (cadena nula).

19.5. Proporcionar funciones recursivas que representen los siguientes conceptos:

a) El producto de dos números naturales.
b) El conjunto de permutaciones de una lista de números.

19.6. Escribir una función recursiva que calcule la función de Ackermann definida de la siguiente forma:

$A(m, n) = n + 1$	si $m = 0$
$A(m, n) = A(m - 1, 1)$	si $n = 0$
$A(m, n) = A(m - 1, A(m, n - 1)$	si $m > 0$, y $n > 0$

19.7. ¿Cuál es la secuencia numérica generada por la función recursiva siguiente?

```
int f(int n)
{
   if (n == 0)
     return 1;
   else if (n == 1)
     return 2;
   else
     return 2*f(n – 2) + f(n – 1);
}
```

19.8. El elemento mayor de un array entero de *n-elementos* se puede calcular recursivamente. Definir la función:

```
int max(int x, int y);
```

que devuelve el mayor de dos enteros *x* e *y*. Definir la función

```
int maxarray(int a[], int n);
```

que utiliza recursión para devolver el elemento mayor de a

Condición de parada: `n == 1`
Incremento recursivo: `maxarray = max (max(a[0]...a [n-2]),a[n-1])`

PROBLEMAS

19.1. La expresión matemática *C(m, n)* en el mundo de la teoría combinatoria de los números, representa el número de combinaciones de *m* elementos tomados de *n* en *n* elementos.

$$C(m, n) = \frac{m!}{n!\,(m - n)!}$$

Escribir y probar una función que calcule *C (m, n)* donde *n!* es el factorial de *n*.

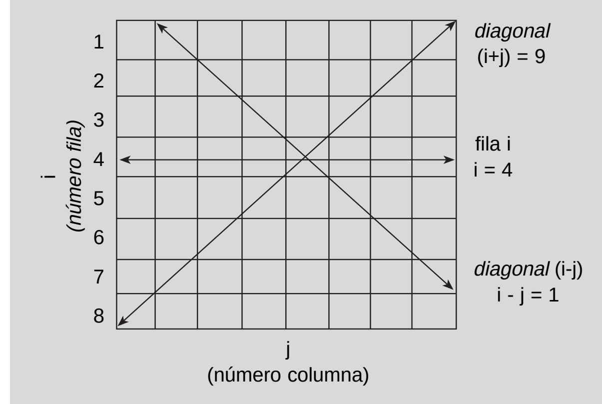
19.2. Un palíndromo es una palabra que se escribe exactamente igual leída en un sentido o en otro. Palabras tales como *level, deed, ala*, etc., son ejemplos de palíndromos. Escribir una función recursiva que devuelva un valor de 1 (verdadero) si una palabra pasada como argumento es un palíndromo y devuelve 0 (falso) en caso contrario.

19.3. Escribir una función recursiva que lista todos los subconjuntos de dos letras para un conjunto dado de letras:

[A, C, E, K] ⇐ [A, C], [A, E], [A, K], [C, E,], [C, K], [E, K]

19.4. El problema de las OCHO REINAS es un famoso problema relativo al ajedrez que tiene por objetivo situar ocho reinas en un tablero de ajedrez, de modo que ninguna sea capaz de atacar a cualquier otra. Una reina se puede mover cualquier número de casillas (cuadrados) verticalmente, horizontalmente o a través de diagonales en el tablero (un cuadriculado de 8 por 8). Escribir un programa que contenga una rutina recursiva que resuelva el problema de las Ocho Reinas.

Sugerencia: Propuesta de numeración de casillas del tablero.



19.5. La suma de los primeros n números enteros responde a la fórmula:

$1 + 2 + 3 + ... + n = n(n + 1) /2$

Inicializar el array A que contiene los primeros 50 enteros. La media de estos elementos del array es entonces 25.5. Comprobar la solución aplicando la función recursiva `media` `(media(float a[], int n))`.

19.6. Desarrollar una función recursiva que cuente el número de números binarios de *n-dígitos* que no tengan dos 1 en una fila. (*Sugerencia*: El número comienza con un 0 o un 1. Si comienza con 0, el número de posibilidades se determina por los restantes *n-1* dígitos. Si comienza con 1, ¿cuál debe ser el siguiente?)

19.7. Leer un número entero positivo *n < 10*. Calcular el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^2 en la forma `x**i`.

Sugerencia:

$(x + 1)^n = C_{n,n}x^n + C_{n,\,n-1}x^{n-1} + C_{n,\,n-2}x^{n-2} + ... + C_{n,\,2}x^2 + C_{n-1}x^1 + C_{n10}x^0$

donde $C_{n,n}$ y $C_{n,0}$ son 1 para cualquier valor de *n*.

La relación de recurrencia de los coeficientes binomiales es:

C(n, 0) = 1
C(n, n) = 1
C(n, k) = C(n-1, k-1) + C(n-1, k)

Estos coeficientes constituyen el famoso Triángulo de Pascal y será preciso definir la función que genera el triángulo

```
1
1  1
1  2     1
1  3     3     1
1  4     6     4     1
```

19.8. Escribir un programa en el que el usuario introduzca 10 enteros positivos y calcule e imprima su factorial.

EJERCICIOS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 373).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

19.1. Explique porqué la siguiente función puede producir un valor incorrecto cuando se ejecute:

```
long factorial (long n)
{
 if (n == 0|| n == 1)
       return 1;
 else
       return n * factorial (--n);
}
```

19.2. ¿Cuál es la secuencia numérica generada por la función recursiva f en el listado siguiente?

```
long f(int n)
{
 if (n == 0|| n == 1)
       return 1;
 else
        return 3 * f(n – 2) + 2 *
  f(n – 1);
 }
```

19.3. Cuál es la secuencia numérica generada por la función recursiva siguiente?

```
int f(int n)
{
 if (n == 0)
       return 1;
 else if (n == 1)
       return 2;
 else
        return 2 * f(n – 2) + f(n
   – 1);
}
```

19.4. Escribir una función que calcule el elemento mayor de un array de n enteros recursivamente.

19.5. Escribir una función recursiva que realice la búsqueda binaria de una clave en un vector ordenado creciente y recursivamente.

PROBLEMAS RESUELTOS EN:

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 374).
2. Sitio web del libro, `www.mhe.es/joyanes` (código fuente).

19.1. Escribir una función que calcule la potencia a^n recursivamente, siendo n positivo.

19.2. Escribir una función recursiva que calcule la función de Ackermann definida de la siguiente forma:

```
A(m, n) = n + 1          si  m = 0
A(m, n) = A (m – 1, 1)  si  n = 0
A(m, n) = A(m – 1,       si   m > 0,
A(m, n – 1))             y n > 0
```

19.3. Escribir una función recursiva que calcule el cociente de la división entera de n entre m, siendo m y n dos números enteros positivos recursivamente.

19.4. Escribir un programa que mediante una función recursiva calcule la suma de los n primeros números pares, siendo n un número positivo.

19.5. Escribir un programa en C++ que mediante recursividad indirecta decida si un número natural positivo es par o impar.

19.6. Escribir una función recursiva para calcular el máximo común divisor de dos números naturales positivos.

19.7. Escribir una función recursiva que lea números enteros positivos ordenados decrecientemente del teclado, elimine los repetidos y los escriba al revés, es decir, ordenado crecientemente. El fin de datos viene dado por el número especial 0.

19.8. Escribir una función iterativa y otra recursiva para calcular el valor aproximado del número *e*, sumando la serie:

$e = 1 + 1/1\,! + 1/2\,! + ... + 1/n\,!$

hasta que los términos adicionales a sumar sean menores que $1.0e^{-8}$.

19.9. Escribir una función recursiva que sume los dígitos de un número natural.

19.10. Escribir una función recursiva, que calcule la suma de los elementos de un vector de reales *v* que sean mayores que un valor *b*.

19.11. Escribir una clase `ListaS` que con la ayuda de una clase `Nodo`, permita la implementación de listas enlazadas recursivas con objetos. Incluir en la clase `ListaS` funciones miembro que permitan insertar recursivamente, y borrar recursivamente, en una lista enlazada ordenada crecientemente, así como mostrar recursivamente una lista.

19.12. Añada a la clase `ListaS` del Problema 19.11 que implementa una lista enlazada ordenada recursivamente con objetos funciones miembro que permitan la inserción y borrado iterativo.

CAPÍTULO 20

Árboles

Contenido

INTRODUCCIÓN

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras *no lineales,* al contrario que los arrays y las listas enlazadas que constituyen *estructuras lineales.*

Los árboles son muy utilizados en informática para representar fórmulas algebraicas, como un método eficiente para búsquedas grandes y complejas en listas dinámicas y en aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de las aplicaciones citadas, los árboles se utilizan en diseño de compiladores, proceso de textos y algoritmos de búsqueda.

En este capítulo se estudiará el concepto de árbol general y los tipos de árboles más usuales: *binario* y *binario de búsqueda*. Asimismo se estudiarán algunas aplicaciones típicas del diseño y construcción de árboles.

CONCEPTOS CLAVE

- Árbol.
- Árbol binario.
- Árbol binario de búsqueda.
- Enorden.
- Nodo.
- Postorden.
- Preorden.
- Subárbol.

20.1. ÁRBOLES GENERALES

Intuitivamente el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general. La Figura 20.1 representa dos ejemplos de árboles generales.

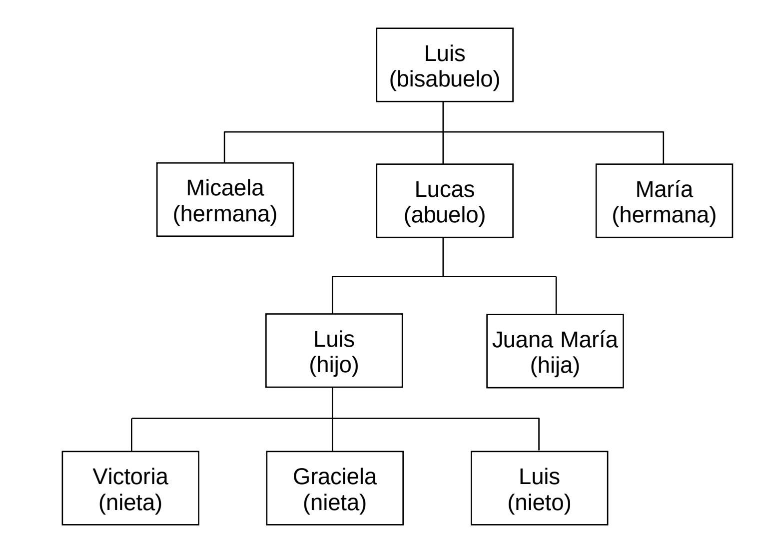


Figura 20.1. Árbol genealógico (bisabuelo-bisnietos).

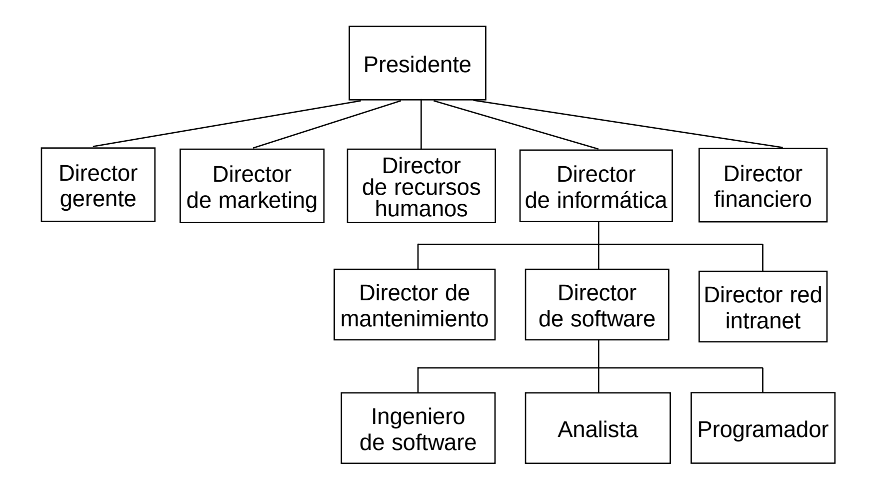


Figura 20.2. Estructura jerárquica tipo árbol.

Un **árbol** consta de un conjunto finito de elementos, denominados **nodos,** y un conjunto finito de líneas dirigidas, denominadas **ramas**, que conectan los nodos. El número de ramas asociado con un nodo es el **grado** del nodo.

Definición 1: Un **árbol** consta de un conjunto finito de elementos, llamados *nodos* y un conjunto finito de líneas dirigidas, llamadas *ramas*, que conectan los nodos.

Definición 2: Un **árbol** es un conjunto de uno o más nodos tales que:

1. Hay un nodo diseñado especialmente llamado **raíz**.

2. Los nodos restantes se dividen en *n* conjuntos disjuntos $T_1...T_n$, en donde cada uno de estos conjuntos es un árbol. A T_1, T_2,...T_n se les denomina *subárboles* del raíz.

Si un árbol no está vacío, entonces el primer nodo se llama **raíz**. Obsérvese en la definición 2 que el árbol ha sido definido de modo recursivo ya que los subárboles se definen como árboles. La Figura 20.3 muestra un árbol.

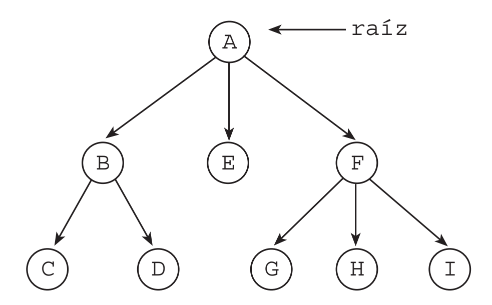


Figura 20.3. Árbol.

20.1.1. Terminología

Además de la raíz existen muchos términos utilizados en la descripción de los atributos de un árbol. En la Figura 20.4, el nodo A es el raíz. Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores.

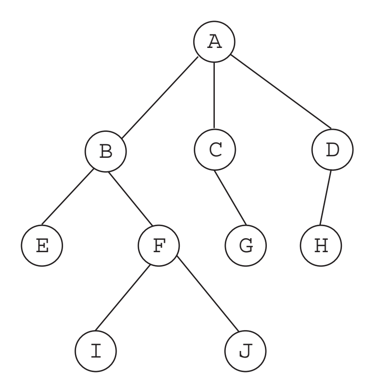


Figura 20.4. Árbol general.

Estos nodos sucesores se llaman **hijos**. Por ejemplo, el nodo `B` es el padre de los hijos `E` y `F`. El padre de `H` es el nodo `D`. Un árbol puede representar diversas generaciones en la familia. Los hijos de un nodo y los hijos de estos hijos se llaman **descendientes** y el padre y abuelos de un nodo son sus **ascendientes**. Por ejemplo, los nodos `E,F,I` y `J` son descendientes de `B`. Cada nodo no raíz tiene un único padre y cada padre tiene cero o más nodos hijos. Dos o más nodos con el mismo padre se llaman **hermanos**. Un nodo sin hijos, tales como `E,I,J,G` y `H,` se llama nodo **hoja**.

El **nivel** de un nodo es su distancia al raíz. El raíz tiene una distancia cero de sí misma, por lo que se dice que el raíz está en el nivel 0. Los hijos del raíz están en el nivel 1, sus hijos están en el nivel 2, y así sucesivamente. Una cosa importante que se aprecia entre los niveles de nodos es la relación entre niveles y hermanos. Los hermanos están siempre al mismo nivel, pero no todos los nodos de un mismo

nivel son necesariamente hermanos. Por ejemplo, en el nivel 2 (Fig. 20.5), `C` y `D` son hermanos, al igual que lo son `G`, `H` e `I`, pero `D` y `G` no son hermanos, ya que ellos tienen diferentes padres.

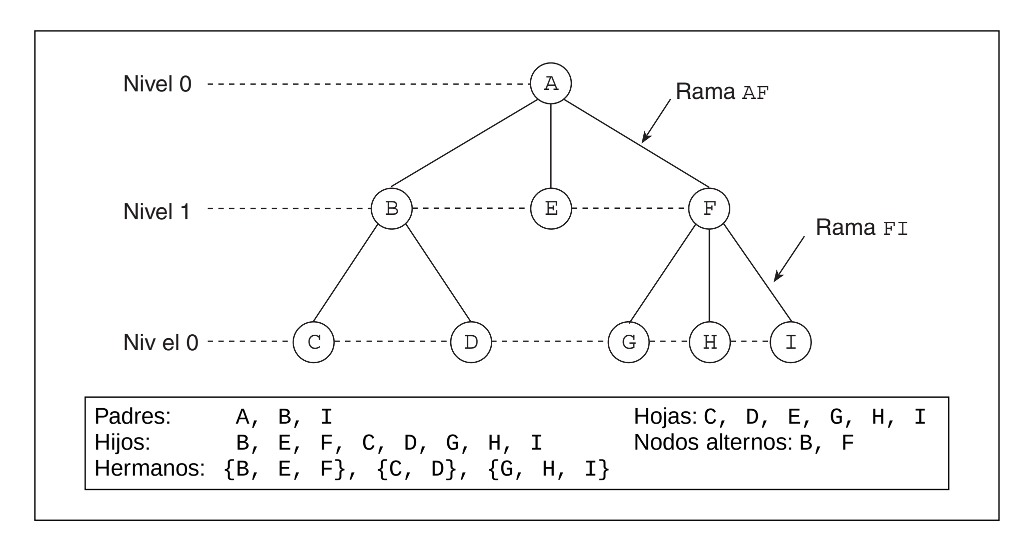


Figura 20.5. Terminología de árboles.

Existen varias formas de dibujar los atributos de los árboles y sus nodos. Un **camino** es una secuencia de nodos en los que cada nodo es adyacente al siguiente. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el raíz. En la Figura 20.5, el camino desde el raíz a la hoja `I` se representa por `AFI`. Incluye dos ramas distintas, `AF` y `FI`.

La altura o profundidad de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. Por definición[1] la altura de un árbol vacío es 0. La Figura 20.5 contiene nodos en tres niveles: 0, 1 y 2. Su altura es 3.

Definición

El nivel de un nodo es su distancia desde la raíz. La altura de un árbol es el nivel de la hoja del camino más largo desde el raíz más uno.

Un árbol se divide en subárboles. Un **subárbol** es cualquier estructura conectada por debajo del raíz. Cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos los descendientes del nodo. El primer nodo de un subárbol se conoce como el raíz del subárbol y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles. En la Figura 20.5, `BCD` es un subárbol al igual que `E` y `FGHI`. Obsérvese que por esta definición, un nodo simple es un subárbol. Por consiguiente, el subárbol `B` se puede dividir en subárboles `C` y `D` mientras que el subárbol `F` contiene los subárboles `G, H` e `I`. Se dice que `G, H, I, C` y `D` son subárboles sin descendientes.

El concepto de subárbol conduce a una *definición recursiva* de un árbol. Un árbol es un conjunto de nodos que:

1. O bien es vacío, o bien
2. Tiene un nodo determinado llamado *raíz* del que jerárquicamente descienden cero o más subárboles, que son también árboles.

---

[1] También se suele definir la **profundidad** de un árbol como el nivel máximo de cada nodo. En consecuencia, la profundidad del nodo raíz es 0, la de su hijo 1, etc. Las dos terminologías son aceptadas.

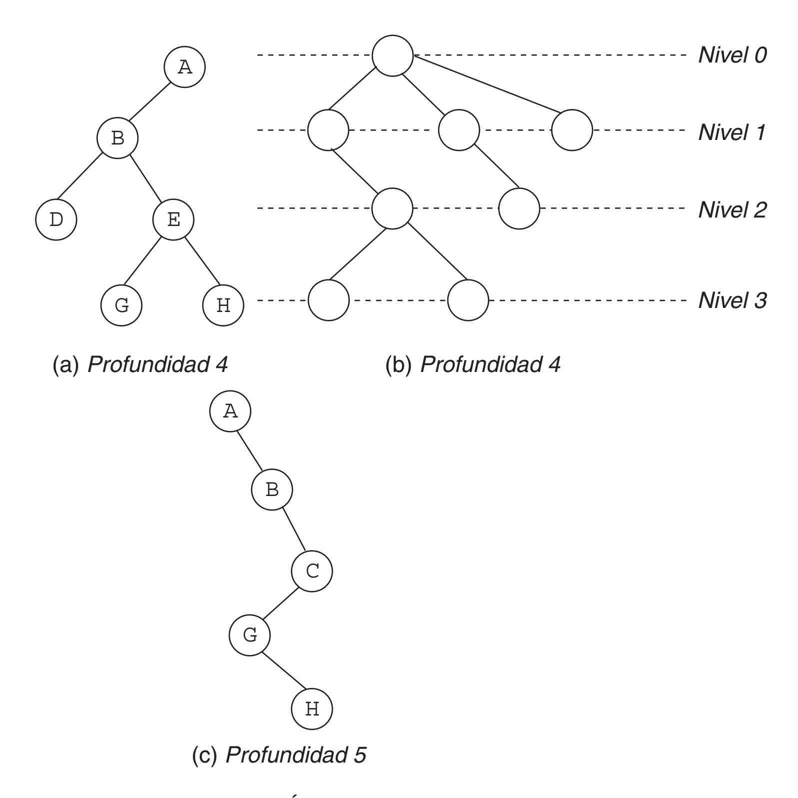


Figura 20.6. Árboles de profundidades diferentes.

Un árbol **está equilibrado** cuando, dado un número máximo de k hijos para cada nodo y la **altura del árbol** h, cada nodo de nivel $l < h - 1$ tiene exactamente k hijos. El árbol está **equilibrado perfectamente** cuando cada nodo de nivel $l < h$ tiene exactamente k hijos.

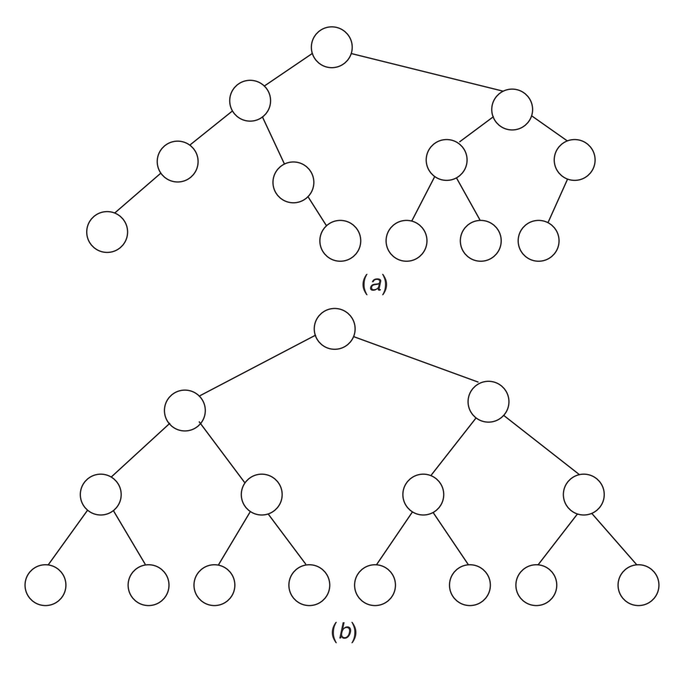


Figura 20.7. (*a*) Un árbol equilibrado. (*b*) Un árbol perfectamente equilibrado.

20.1.2. Representación de un árbol

Aunque un árbol se implementa en un lenguaje de programación como C++ mediante punteros, cuando se ha de representar en papel, existen tres formas diferentes de representación. La primera es el diagrama o carta de organización utilizada hasta ahora en las diferentes figuras. El término que se utiliza para esta notación es el de árbol general.

Representación en niveles de profundidad

Este tipo de representación es el utilizado para representar sistemas jerárquicos en modo texto o número, en situaciones tales como facturación, gestión de *stocks* en almacenes, etc.

Por ejemplo, en las Figuras 20.8 y 20.9 se aprecia una descomposición de una computadora en sus diversos componentes en un árbol. Otro ejemplo podría ser una distribución en árbol de las piezas de una tienda de recambios de automóviles distribuidas en niveles de profundidad según los números de parte o códigos de cada repuesto (motor, bujía, batería, piloto, faro, embellecedor, etc.).

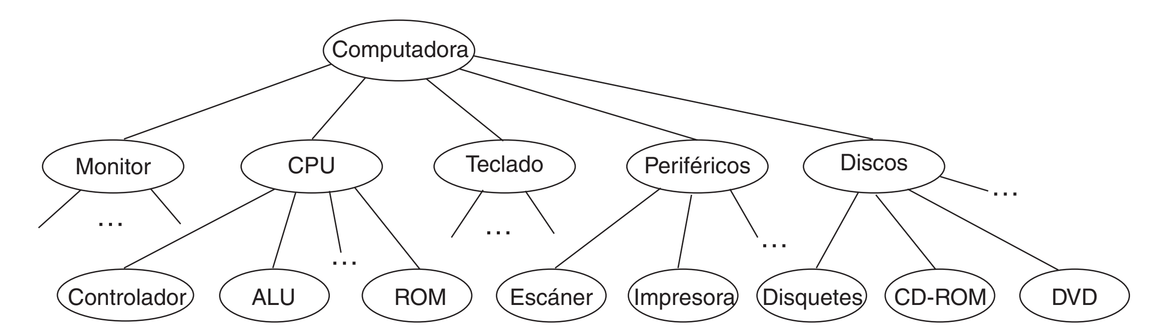


Figura 20.8. Árbol general (computadora).

Número código	Descripción
501	Computadora
501-11	Monitor
...	
501-21	CPU
501-211	Controlador
501-212	ALU
...	...
501-219	ROM
501-31	Teclado
...	...
501-41	Periféricos
501-411	Escáner
501-412	Impresora
501-51	Discos
501-511	Disquetes (3½")
501-512	CD-ROM
501-513	DVD

Figura 20.9. Árbol en nivel de profundidad (computadora).

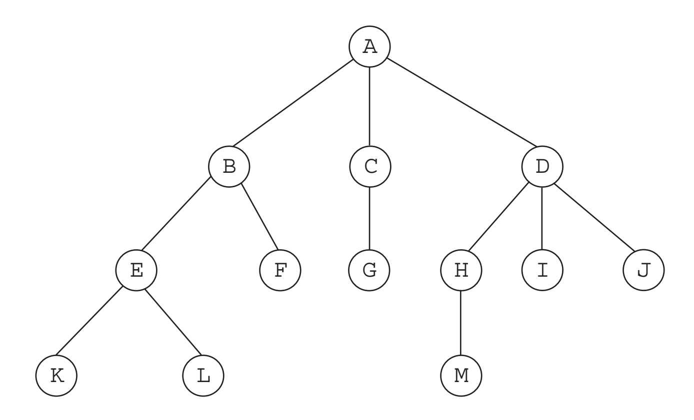
Representación de lista

Otro formato utilizado para representar un árbol es la lista entre paréntesis. Ésta es la notación utilizada con expresiones algebraicas. En esta representación, cada paréntesis abierto indica el comienzo de un nuevo nivel; cada paréntesis cerrado completa un nivel y se mueve hacia arriba un nivel en el árbol. La notación en paréntesis de la Figura 20.3 es:

```
A (B (C, D), E, F, (G, H, I))
```

Ejemplo 20.1

Convertir el árbol general siguiente en representación en lista.



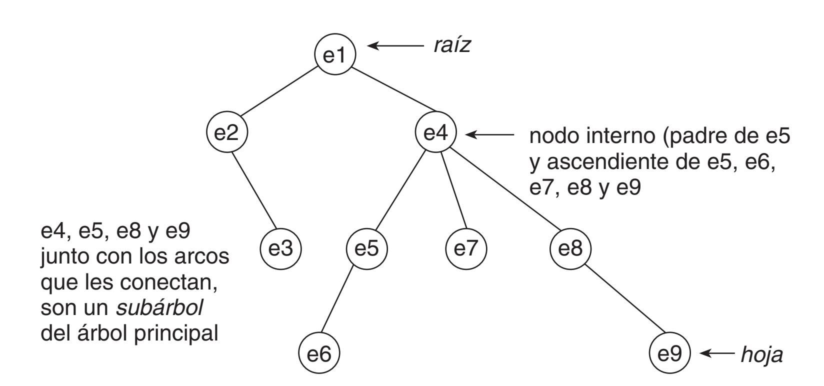
La solución es `A(B(E(K, L), F), C(G), D(H(M), I, J)))`

20.2. RESUMEN DE DEFINICIONES

1. Dado un conjunto E de elementos,
   - Un árbol puede estar *vacío*; es decir, no contiene ningún elemento.
   - Un árbol *no vacío* puede constar de un único elemento $e \equiv E$ denominado **nodo**.
   - Un árbol consta de un nodo $e \equiv E$, conectado por arcos directos a un número finito de otros árboles.

2. Definiciones
   - El primer nodo de un árbol, normalmente dibujado en la posición superior, se denomina **raíz** del árbol.
   - Las flechas que conectan un nodo a otro se llaman **arcos** o **ramas**.
   - Los **nodos terminales**, esto es, nodos de los cuales no se deduce ningún nodo, se denominan **hojas**.
   - Los nodos que no son hojas se denominan **nodos internos** o **nodos no terminales**.
   - En un árbol una rama va de un nodo n_1 a un nodo n_2, se dice que n_1 es el padre de n_2 y que n_2 es un **hijo** de n_1.
   - n_1 se llama **ascendiente** de n_2 si n_1 es el padre de n_2 o si n_1 es el padre de un ascendiente de n_2.
   - n_2 se llama **descendiente** de n_1 si n_1 es un ascendiente de n_2.
   - Un **camino** de n_1 a n_2 es una secuencia de arcos contiguos que van de n_1 a n_2.
   - La **longitud** de un camino es el número de arcos que contiene (en otras palabras el número de nodos –1).
   - El **nivel** de un nodo es la longitud del camino que lo conecta al raíz.

- La **profundidad** o **altura** de un árbol es la longitud del camino más largo que conecta el raíz a una hoja más 1.
- Un **subárbol** de un árbol es un subconjunto de nodos del árbol, conectados por ramas del propio árbol, esto es, a su vez un árbol.
- Sea S un subárbol de un árbol A: si para cada nodo *n* de SA, SA contiene también todos los descendientes de *n* en A. SA se llama un **subárbol completo** de A.
- Un árbol está **equilibrado** cuando, dado un número máximo *K* de hijos de cada nodo y la **altura del árbol** *h*, cada nodo de nivel *l* < *h-1* tiene exactamente *K* hijos. El árbol está equilibrado perfectamente entre cada nodo de nivel *l*<*h* cuando tiene exactamente *K* hijos.



20.3. ÁRBOLES BINARIOS

Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario, cada nodo puede tener cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como *hijo izquierdo* y el nodo de la derecha como *hijo derecho*. (Véase la Figura 20.10.)

> ***Nota***
>
> Un árbol binario no puede tener más de dos subárboles.

Un árbol binario es una estructura recursiva. Cada nodo es el raíz de su propio subárbol y tiene hijos, que son raíces de árboles llamados los subárboles derecho e izquierdo del nodo, respectivamente. Un árbol binario se divide en tres subconjuntos disjuntos (véase la Figura 20.11):

{R}	Nodo raíz.
$\{I_1, I_2, ... In\}$	Subárbol izquierdo de R.
$\{D_1, D_2, ... D_n\}$	Subárbol derecho de R.

En cualquier nivel *n*, un árbol binario puede contener de 1 a 2^n nodos. El número de nodos por nivel contribuye a la densidad del árbol.

En la Figura 20.12 (*a*) el árbol A contiene 8 nodos en una profundidad de 3, mientras que b contiene 5 nodos profundidad 4. Este último caso es una forma especial, denominado **árbol degenerado**, en el que existe un solo nodo hoja (E) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

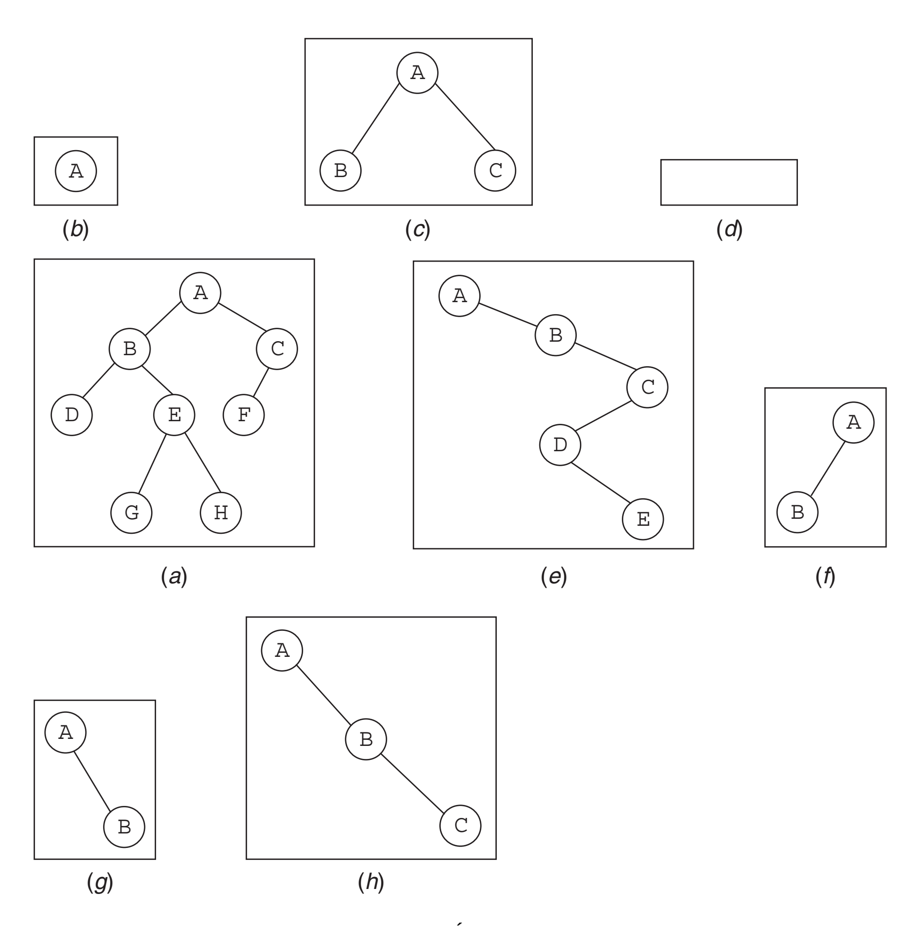


Figura 20.10. Árboles binarios.

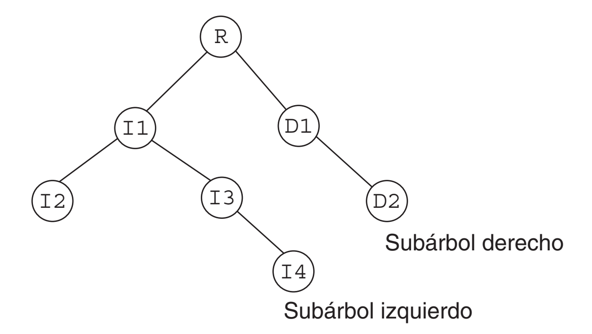


Figura 20.11. Árbol binario.

20.3.1. Equilibrio

La distancia de un nodo al raíz determina la eficiencia con la que puede ser localizado. Por ejemplo, dado cualquier nodo de un árbol, a sus hijos se puede acceder siguiendo sólo un camino de bifurcación o de ramas, el que conduce al nodo deseado. De modo similar, los nodos a nivel 2 de un árbol sólo pueden ser accedidos siguiendo sólo dos ramas del árbol.

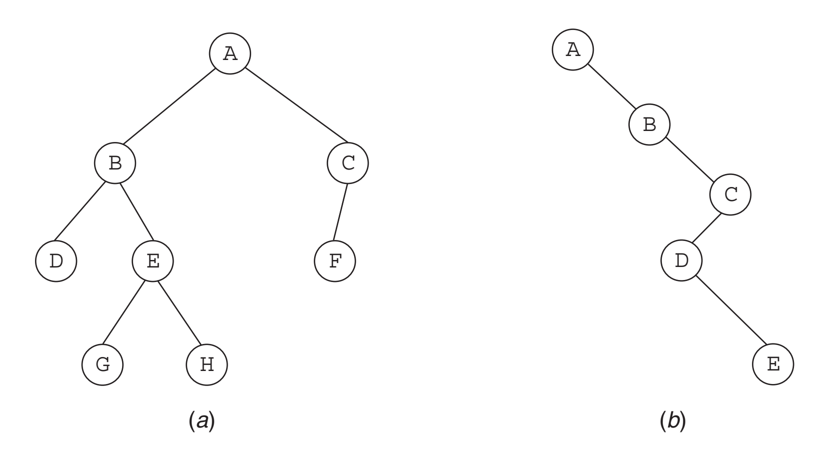


Figura 20.12. Árboles binarios: (*a*) profundidad 4; (*b*) profundidad 5.

La característica anterior nos conduce a una característica muy importante de un árbol binario, su **balance** o **equilibrio**. Para determinar si un árbol está equilibrado, se calcula su factor de equilibrio. El **factor de equilibrio** de un árbol binario es la diferencia en altura entre los subárboles izquierdo y derecho. Si definimos la altura del subárbol izquierdo como H_I y la altura del subárbol derecho como H_D, entonces el factor de equilibrio del árbol `B` se determina por la siguiente fórmula:

$$B = H_D - H_I$$

Utilizando esta fórmula el equilibrio de los ocho árboles de la Figura 20.10 son (*a*) 0, (*c*) 0, (*d*) 0, (*f*) –1, (*g*) 1, (*e*) 4, (*h*) 2.

Un árbol está **equilibrado** si su equilibrio o balance es *cero* y sus subárboles son también equilibrados. Dado que esta definición ocurre raramente, se aplica una definición alternativa. Un árbol binario está equilibrado si la altura de sus subárboles difiere en no más de uno (su factor de equilibrio es `-1, 0, +1`) y sus subárboles son también equilibrados (véase la Figura 2.13).

20.3.2. Árboles binarios completos

Un árbol binario **completo** tiene el máximo número de entradas para su altura. Esto sucede cuando el último nivel está lleno.

Un árbol binario completo de profundidad *n* es un árbol en el que cada nivel *0* a nivel *n-1* tiene un conjunto lleno de nodos y todos los nodos hoja a nivel *n* ocupan las posiciones más a la izquierda del árbol. Un árbol binario completo que contiene 2^n nodos a nivel *n* es un **árbol lleno**. Un árbol lleno es un árbol binario en el que cada nodo no hoja tiene dos hijos. La Figura 20.13 muestra un árbol binario completo y uno lleno.

El último caso de árbol es un tipo especial denominado **árbol degenerado** en el que hay un solo nodo hoja (`E`) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

Un árbol binario se considera **casi completo** si tiene la misma altura para sus nodos del mismo nivel y todos los nodos del último nivel se encuentran a la izquierda. Los árboles completos y casi completos se muestran en la Figura 20.14.

Los árboles binarios y completos de profundidad *n* proporcionan algunos datos matemáticos que es necesario comentar. En cada caso, existe un nodo (2^0) al nivel 0 (raíz), dos nodos (2^1) a nivel 1, cuatro nodos (2^2) a nivel 2, etc. A través de los primeros *k*-1 niveles hay 2^k-1 nodos.

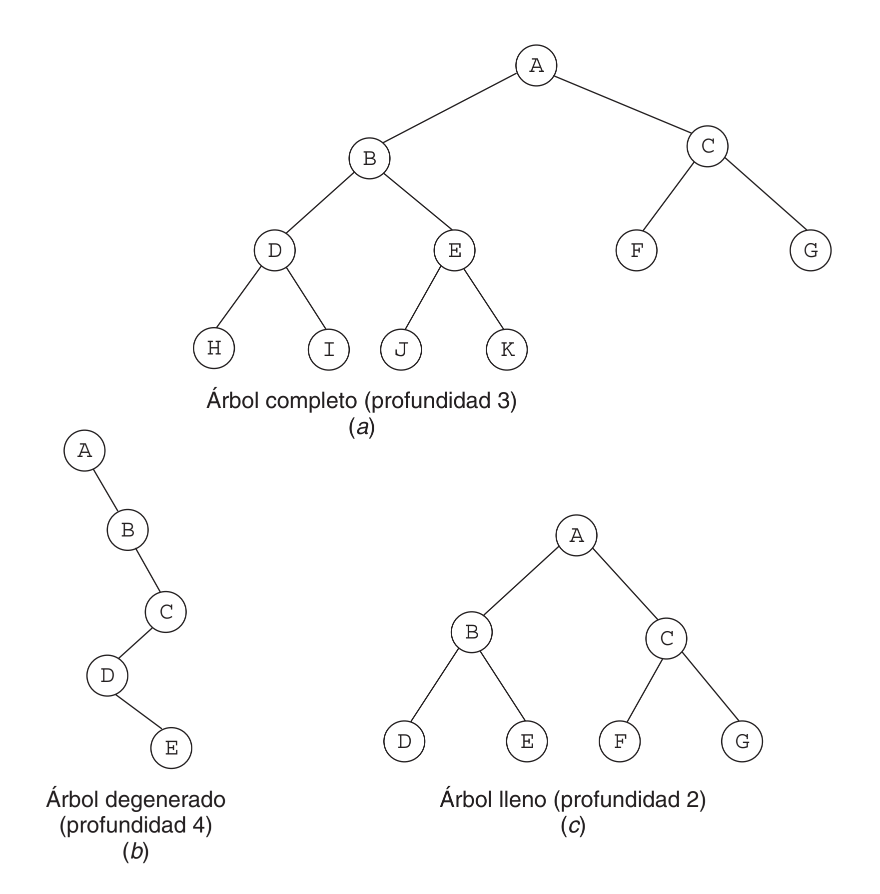


Figura 20.13. Clasificación de árboles binarios: (*a*) completo; (*b*) degenerado; (*c*) lleno.

$1 + 2 + 4 + ... + 2^{k-1} = 2^k - 1$

A nivel *k*, el número de nodos adicionados están en el rango de un mínimo de 1 a un máximo de 2^k (lleno). Con un árbol lleno, el número de nodos es

$1 + 2 + 4 + ... + 2^{k-1} + 2^k = 2^{k+1} -1$

El número de nodos *n* en un árbol binario completo cumple la inigualdad

$2^k \Im n \Im 2^{k+1} -1 < 2^{k+1}$

Aplicando logaritmos a la ecuación con desigualdad anterior

$k \Im log_2 (n) < k + 1$

Por ejemplo, un árbol completo de profundidad 3 tiene $2^4 -1 = 15$ nodos.

(*a*)

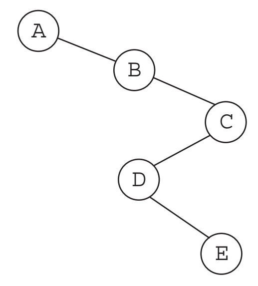
(*b*)

Figura 20.14. (*a*) Árboles completos (en niveles 0, 1 y 2); (*b*) árboles casi completos (en nivel 2).

Ejemplo 20.2

Calcular la profundidad máxima y mínima de un árbol con 5 nodos.

La profundidad máxima de un árbol con 5 nodos es 5.



La profundidad mínima k de un árbol con 5 nodos es

$k \Im \log_2 (5) < k + 1$

$log_2 (5) = 2.32$ y $K = 2 + 1 = 3$

Ejemplo 20.3

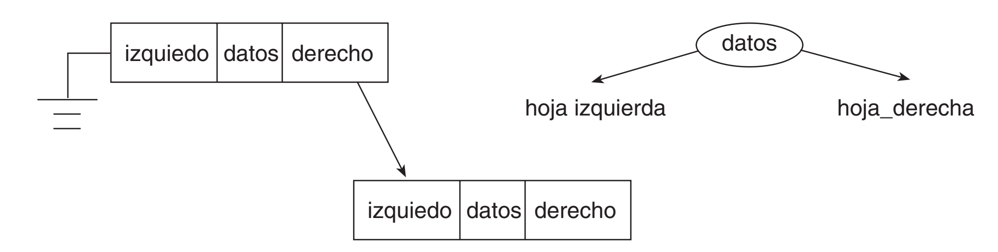
La profundidad de un árbol degenerado con n nodos es n – 1, dado que es la longitud del camino más largo (raíz a nodo) más 1.

En un árbol binario completo con *n* nodos, la profundidad del árbol es el valor entero de $log_2\ n$, que es, a su vez, la distancia del camino más largo desde el raíz a un nodo. Suponiendo que el árbol tiene *n = 10.000* elementos, el camino más largo es

```
int (long2 10000) + 1 = int (13.28) + 1 = 13 14
```

20.4. ESTRUCTURA DE UN ÁRBOL BINARIO

La estructura de un árbol binario se construye con nodos. Cada nodo debe contener el campo datos (datos a almacenar) y dos campos punteros, uno al subárbol izquierdo y otro al subárbol derecho, que se conocen como **puntero izquierdo (izquierdo, izdo)** y **puntero derecho (derecho, dcho)** respectivamente. Un valor `null` indica un árbol vacío



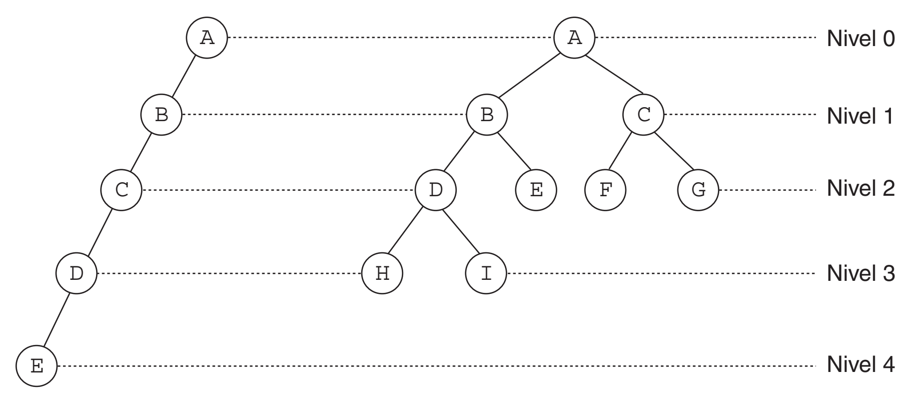
El algoritmo correspondiente a la estructura de un árbol es el siguiente

```
Nodo
   subarbolIzquierdo      < puntero a Nodo>
   datos                  < TipoDato >
   subarbolDerecho        < puntero a Nodo>
Fin Nodo
```

La Figura 20.15 muestra un árbol binario y su estructura en nodos.

Ejemplo 20.4

Representar la estructura en nodo de los dos árboles binarios A.



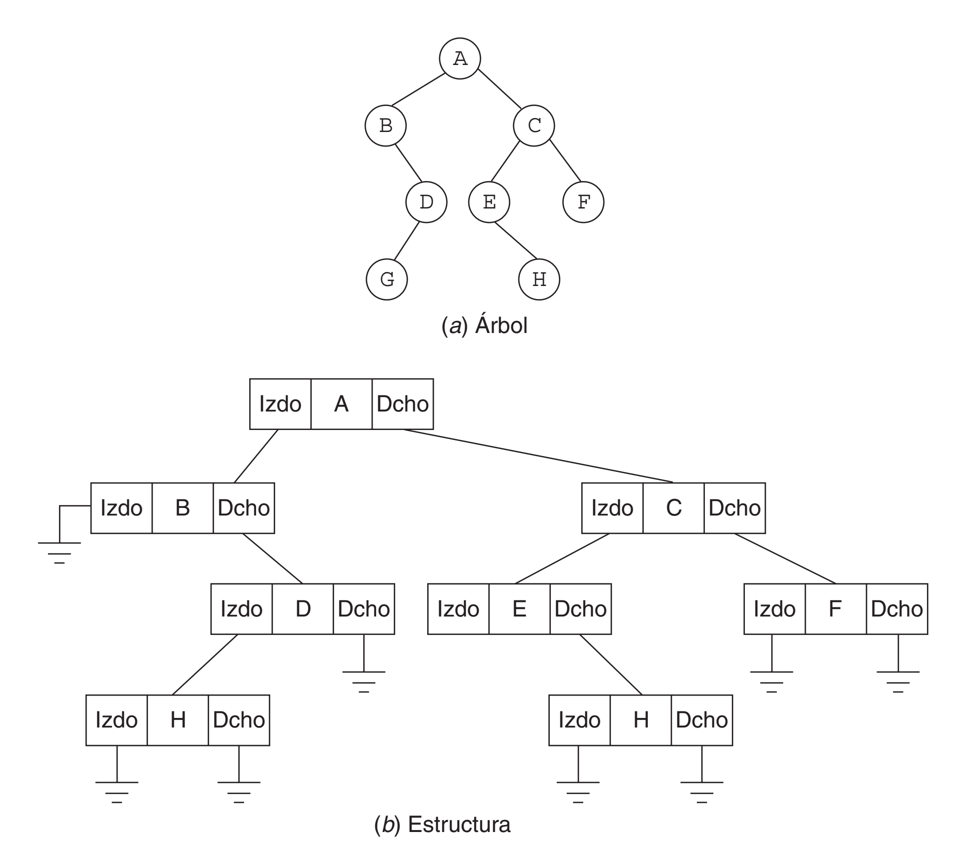


Figura 20.15. Árbol binario y su estructura en nodos.

La representación enlazada de los árboles binarios A es:

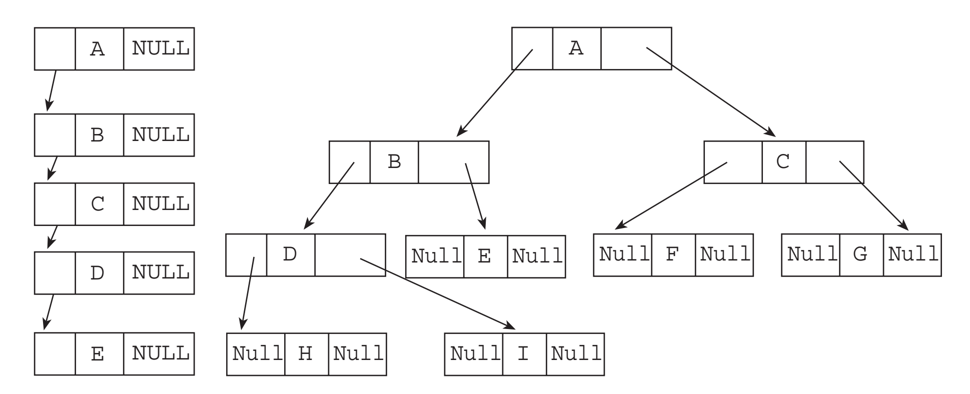
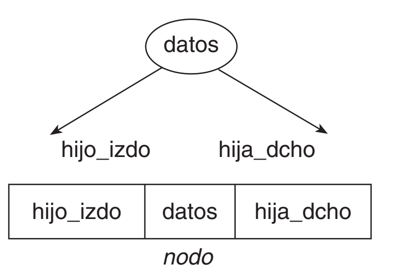


Figura 20.16. Representación enlazada de árboles binarios.

20.4.1. Diferentes tipos de representaciones en C++

Los nodos pueden ser representados con las estructuras `struct` o `class`; en el primer caso se tiene la ventaja de conservar la compatibilidad con el lenguaje C y en el segundo caso se aprovechan las características propias de C++. Suponiendo que el nodo tiene los campos `datos, Hijo_izdo` e `Hijo_dcho`.

Representación 1 (estilo C)




```
typedef struct nodo *puntero_arbol;
struct nodo {
   int datos;
   puntero_arbol hijo_izdo, hijo_dcho;
};
```

Representación 2 (estilo C)

```
struct Nodo {
   TipoElemento Info;
   struct Nodo *Hijo_izdo, *Hijo_dcho;
};

typedef struct Nodo ElementoDeArbolBin;
typedef ElementoDeArbolBin *ArbolBinario;
```

Representación 3 (estilo C++)

```
class arbol {
   class nodo {
      char *datos;
      nodo *derecho;
      nodo *izquierdo;
   friend class arbol;
   };
public:
   nodo *raiz;      // cima del árbol (raíz)

   arbol(void)  { raiz = NULL; };
   // ... otras funciones miembro
};
```

Representación 4 (estilo C++ con `templates`)

```
//Arbol binario
template <class T>
class ArbolBin;

// declara un objeto nodo árbol de un árbol binario
template <class T>
class NodoArbol
{
   private:
      // apunta a los hijos izquierdo y derecho del nodo
      NodoArbol <T> *izquierdo;
      NodoArbol <T> *derecho;

   public:
      // miembro público, se puede actualizar su valor
      T datos;
```

```
      // constructor
      NodoArbol (const T& item, NodoArbol <T> *ptri = NULL,
                                NodoArbol <T> *ptrd = NULL);
      // métodos de acceso a los campos puntero
      NodoArbol <T>* Izquierdo(void) const;
      NodoArbol <T>* Derecho(void) const;

      // hacer a ArbolBin un amigo ya que necesita acceder a los
      // campos puntero izquierdo y derecho del nodo
      friend class ArbolBin <T>;
};
```

La descripción de la clase `ArbolBin` es como sigue. El constructor inicializa los datos y los campos puntero. El uso del puntero por omisión `NULL` hace que el nodo se inicialice como un nodo hoja. Con un puntero `P` de `NodoArbol` que se pasa como parámetro, el constructor conecta `P` como hijo izquierdo o derecho del nuevo nodo. Los métodos de acceso `Izquierdo` y `Derecho` devuelven el valor correspondiente del puntero. La clase `ArbolBin` se declara como amiga de `NodoArbol` y puede modificar los punteros. Otros clientes deben utilizar el constructor para crear los punteros y, a continuación, utilizar métodos `Izquierdo` y `Derecho` para un recorrido del árbol.

20.4.2. Operaciones en árboles binarios

Algunas de las operaciones típicas que se realizan en árboles binarios son:

- Determinar su altura.
- Determinar su número de elementos.
- Hacer una copia.
- Visualizar el árbol binario en pantalla o en impresora.
- Determinar si dos árboles binarios son idénticos.
- Borrar (eliminar el árbol).
- Si es un árbol de expresión[2], evaluar la expresión.
- Si es un árbol de expresión, obtener la forma de paréntesis de la expresión.

Todas estas operaciones se pueden realizar recorriendo el árbol binario de un modo sistemático. El recorrido de un árbol es la operación de vista al árbol, o lo que es lo mismo, la visita a cada nodo del árbol una vez y sólo una. La visita de un árbol es necesaria en muchas ocasiones, por ejemplo, si se desea imprimir la información contenida en cada nodo. Existen diferentes formas de visitar o recorrer un árbol que se estudiarán más tarde.

20.4.3. Estructura y representación de un árbol binario

La estructura de un árbol binario es aquella que en cada nodo se almacena un dato y su hijo izquierdo e hijo derecho. En C++ puede representarse de la siguiente forma.

1. La clase nodo se encuentra anidada dentro de la clase árbol.

```
class arbol
{
```

[2] En el apartado siguiente se estudia el importante concepto de *árbol de expresión*.

```
  class nodo
   {
     public:
        char *datos;
     private:
        nodo *derecho;
        nodo *izquierdo;
     friend class arbol;
   };
public:
    nodo *raiz;                               // raíz del árbol (raíz)
     arbol()  {raiz = NULL; };
     // ... otras funciones miembro
};
```

2. La clase `ArbolBin` no tiene dentro la clase `NodoArbol`, pero es declarada como clase amiga. Se realiza además con tipos genéricos.

```
template <class T>
class ArbolBin;                         // aviso a la clase NodoArbol.

// declara un objeto nodo árbol de un árbol binario
template <class T>
class NodoArbol
{
     private:
        // apunta a los hijos izquierdo y derecho del nodo
        NodoArbol <T> *izquierdo;
        NodoArbol <T> *derecho;
        T datos;
     public:

        // constructor
        NodoArbol (T item, NodoArbol <T> *ptri,NodoArbol <T> *ptrd)
        {
          datos  = item;
          izquierdo = ptri;
          derecho = ptrd;
        }

                          // métodos de acceso a los atributos puntero
        NodoArbol <T>* OIzquierdo() {return izquierdo;}
        NodoArbol <T>* ODerecho(void) {return derecho;};
        void PIzquierdo(NodoArbol <T>* Izq) {izquierdo = Izq;}
        void PDerecho(NodoArbol <T>* Drc) {derecho = Drc;}

                                 // métodos de acceso al atributo dato
        T Odatos() { return datos;}
        void Pdatos( T e){ datos = e;}
            // hacer a ArbolBin un amigo ya que necesita acceder a los
                       // campos  puntero izquierdo y derecho del nodo
        friend class ArbolBin <T>;
};
```

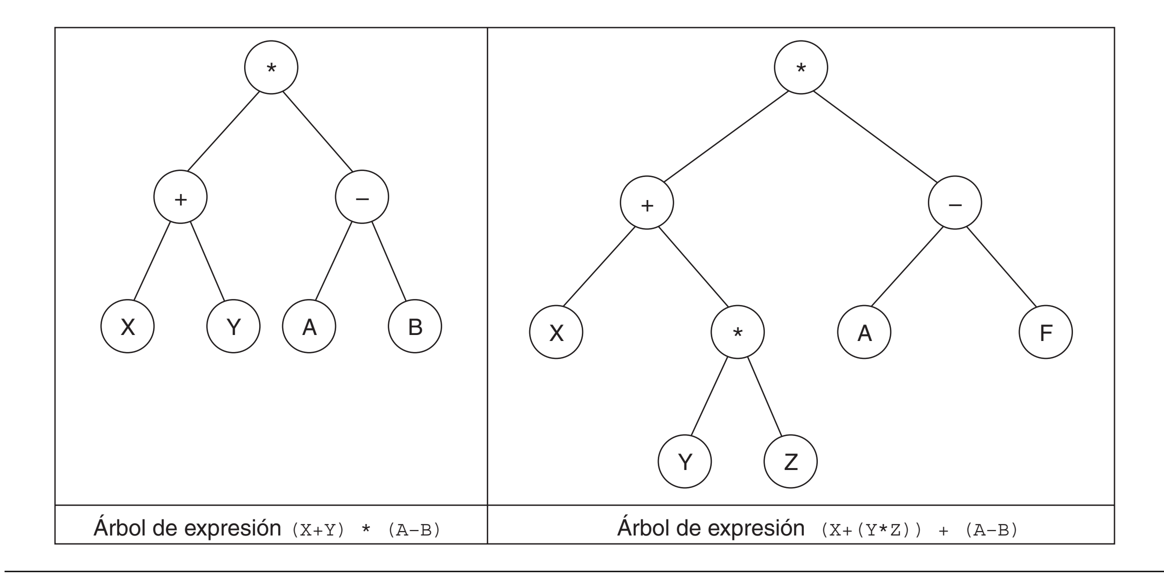
```
template <class T>
class ArbolBin
{
private:
    NodoArbol <T> * p;
public:
    ArbolBin() { p = NULL;}
                                            // otras funciones miembro
} ;
```

Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

1. Cada hoja es un operando.
2. Los nodos raíz e internos son operadores.
3. Los subárboles son subexpresiones en las que el nodo raíz es un operador.

Ejemplo 20.5

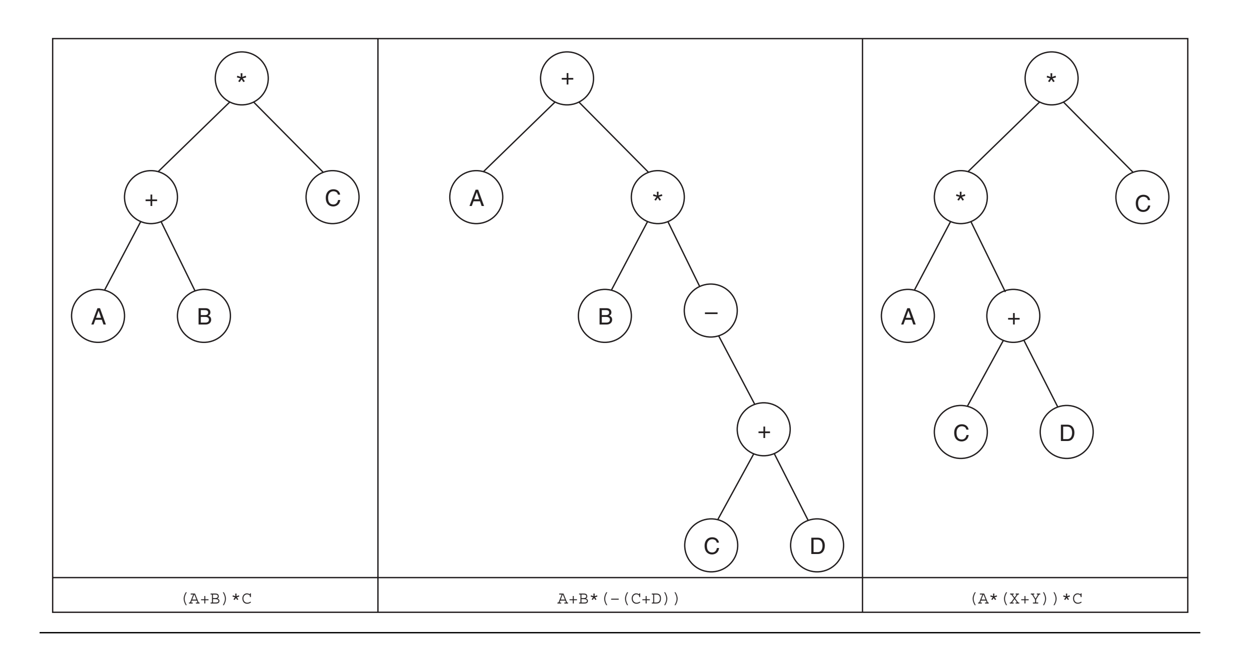
Árboles de expresiones.



Ejemplo 20.6

Árboles de expresiones asociados a:

```
(A+B)*C,  A+B*(-(C+D))  (A*(X+Y))*C
```



20.5. ÁRBOLES DE EXPRESIÓN

Una aplicación muy importante de los árboles binarios son los *árboles de expresión*. Una **expresión** es una secuencia de *tokens* (componentes de léxicos) que siguen unas reglas prescritas. Un *token* puede ser o bien un operando o bien un operador.

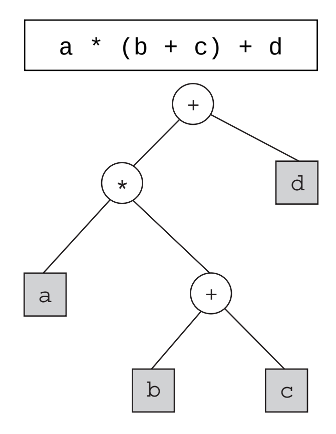


Figura 20.17. Una expresión infija y su árbol de expresión.

Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

1. Cada hoja es un operando.
2. Los nodos raíz e internos son operadores
3. Los subárboles son subexpresiones en las que el nodo raíz es un operador.

La Figura 20.17 representa una expresión infija y su árbol de expresión.

Los árboles binarios se utilizan para representar expresiones en memoria; esencialmente, en compiladores de lenguaje de programación. La Figura 20.18 muestra un árbol binario de expresiones para la expresión aritmética `(a + b)* c`.

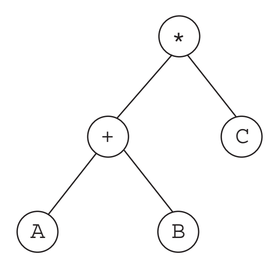


Figura 20.18. Árbol binario de expresiones que representa `(A+B) *C`.

Obsérvese, que los paréntesis no se almacenan en el árbol pero están implicados en la forma del árbol. Si se supone que todos los operadores tienen dos operandos, se puede representar una expresión por un árbol binario cuya raíz contiene un operador y cuyos subárboles izquierdo y derecho son los operandos izquierdo y derecho respectivamente. Cada operando puede ser una letra (`X,Y,A,B`, etc.) o una subexpresión representada como un subárbol. En la Figura 20.19 se puede ver cómo el operador que está en la raíz es `*`, su subárbol izquierdo representa la subexpresión `(x + y)` y su subárbol derecho representa la subexpresión `(A-B)`. El nodo raíz del subárbol izquierdo contiene el operador (+) de la subexpresión izquierda y el nodo raíz del subárbol derecho contiene el operador (-) de la subexpresión derecha. Todos los operandos letras se almacenan en nodos hojas.

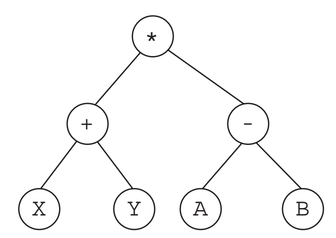


Figura 20.19. Árbol de expresión `(x+y) * (A-B)`.

Utilizando el razonamiento anterior, se puede escribir la expresión almacenada como

```
(X+(Y*Z))*(A-B)
```

en donde se han insertado paréntesis alrededor de subexpresiones del árbol (los operandos `Y, Z` de la subexpresión más interna tienen el nivel mayor).

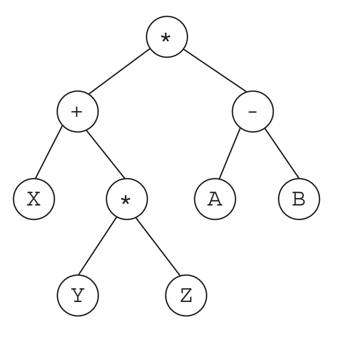
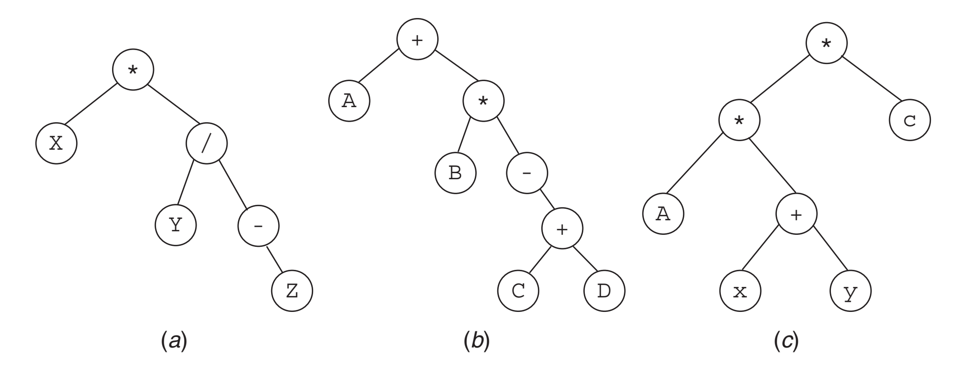


Figura 20.20. Árbol de expresión `(X+(Y*Z))*(A-B)`.

Ejemplo 20.7

Deducir las expresiones que representan los siguientes árboles binarios:



Soluciones

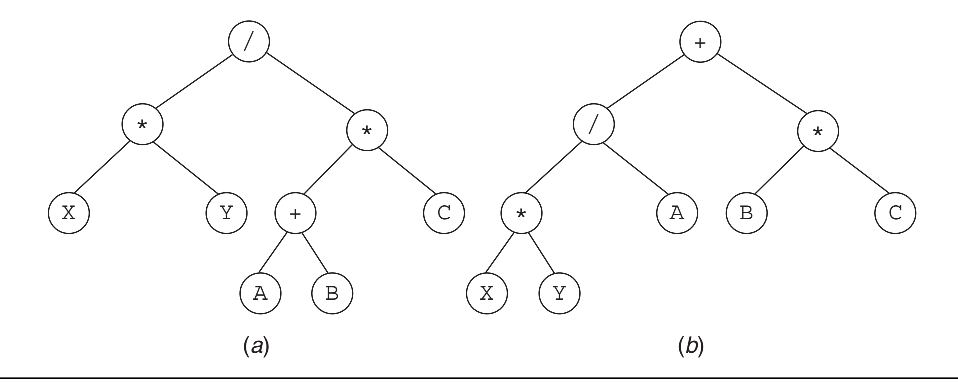
a) `X*(Y/-Z)`
b) `A +(B*-(C+D))`
c) `(A*(X+Y))*C`

Ejemplo 20.8

Dibujar la representación en árbol binario de cada una de las siguientes expresiones:

a) `X*Y/(A+B)*C`
b) `X*Y/A+B*C`

Soluciones



20.5.1. Reglas para la construcción de árboles de expresión

Los árboles de expresiones se utilizan en las computadoras para evaluar expresiones usadas en programas. El algoritmo más sencillo para construir un árbol de expresión es aquel que lee una expresión completa que contiene paréntesis en la misma. Una *expresión con paréntesis* es aquélla en que

1. La prioridad se determina sólo por paréntesis.
2. La expresión completa se sitúa entre paréntesis.

Por consiguiente, `(4+(5*6))` es un ejemplo de una expresión completa entre paréntesis. Su valor es `34`. Si se desea cambiar las prioridades, se escribe `((4+5)*6)`, su valor es `54`. A fin de ver la prioridad en las expresiones, considérese la expresión

```
(4*5) + 6/7 - (8+9)
```

Los operadores con prioridad más alta son * y /; es decir,

```
(4*5) + (6/7) - (8+9)
```

El orden de los operadores en este caso es + y –. Por consiguiente, se puede escribir

```
((4*5) + (6/7)) - (8+9)
```

Por último, la expresión completa entre paréntesis será

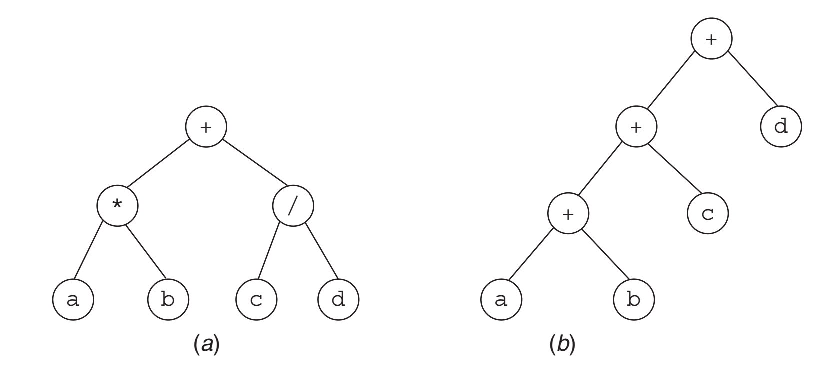
```
(((4*5) + (6/7)) - (8+9))
```

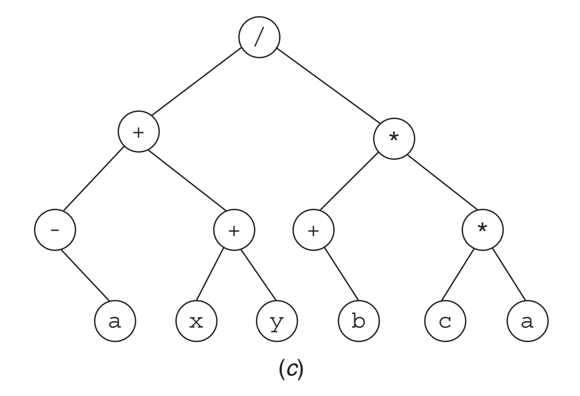
El algoritmo para la construcción de un árbol de expresión es:

1. La primera vez que se encuentra un paréntesis a la izquierda, crea un nodo y lo hace en el raíz. A éste se le llama *nodo actual* y se sitúa su puntero en una pila.
2. Cada vez que se encuentre un nuevo paréntesis a la izquierda, crear un nuevo nodo. Si el nodo actual no tiene un hijo izquierdo, hacer al nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho. Hacer al nuevo nodo el nodo actual y situar su puntero en una pila.
3. Cuando se encuentra un operando, crear un nuevo nodo y asignar el operando a su campo de datos. Si el nodo actual no tiene un hijo izquierdo, hacer al nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho.
4. Cuando se encuentra un operador, sacar un puntero de la pila y situar el operador en el campo datos del nodo del puntero.
5. Ignorar paréntesis derecho y blancos.

Ejemplo 20.9

Calcular las expresiones correspondientes de los árboles de expresión





(*c*)

Las soluciones correspondientes son:

a. `(a * b)  + ( c / d)` ***c.*** `((-a)+(x+y))/((+b)*(c*a))`
b. `((a + b) + c) + d`

Ejercicio 20.1 (a realizar por el lector)

Dibujar los árboles binarios de expresión correspondientes a cada una de las siguientes expresiones:

a) `(a + b) / (c - d * e) + e + g * h/a`
b) `-x -y * z + (a + b + c / d * e)`
c) `((a + b) > (c - e)) || a < f && (x < y || y > z)`

20.6. RECORRIDO DE UN ÁRBOL

Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario que las listas enlazadas, los árboles binarios no tienen realmente un primer valor, un segundo valor, un tercer valor, etc. ¿Se puede afirmar que el raíz viene el primero, pero quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayoría de las aplicaciones binarias son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

Un **recorrido de un árbol binario** requiere que cada nodo del árbol sea procesado (visitado) una vez y sólo una en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, profundidad y anchura.

En el **recorrido en profundidad**, el proceso exige un camino desde el raíz a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo.

En el **recorrido en anchura**, el proceso se realiza horizontalmente desde el raíz a todos sus hijos, a continuación a los hijos de sus hijos y así sucesivamente hasta que todos los nodos han sido procesados. En otras palabras, en el recorrido en anchura cada nivel se procesa totalmente antes de que comience el siguiente nivel.

> El **recorrido** de un árbol supone visitar cada nodo sólo una vez.

Dado un árbol binario que consta de un raíz, un subárbol izquierdo y un subárbol derecho se pueden definir tres tipos de secuencia de recorrido. Estos recorridos estándar se muestran en la Figura 20.21.

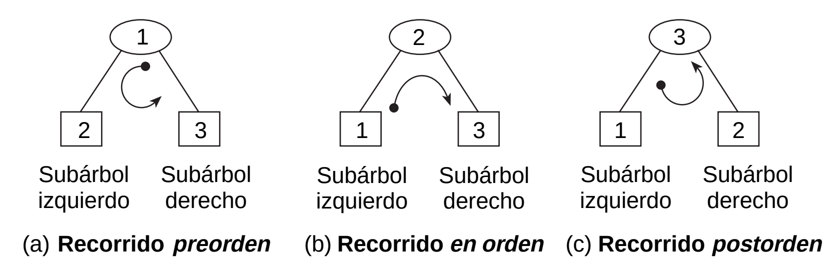


Figura 20.21. Recorridos de árboles binarios.

La designación tradicional de los recorridos utiliza un nombre para el nodo raíz (`N`), para el subárbol izquierdo (`I`) y para el subárbol derecho (`D`).

Según sea la estrategia a seguir, los recorridos se conocen como ***enorden*** (*inorder*), ***preorden*** (*preorder*) y ***postorden*** (*postorder*).

Preorden (nodo-izquierdo-derecho) (`NID`).
Enorden (izquierdo-nodo-derecho) (`IND`).
Postorden (izquierdo-derecho-nodo) (`IDN`).

20.6.1. Recorrido *preorden*

El recorrido preorden[3] (`NID`) conlleva los siguientes pasos, en los que el raíz va antes que los subárboles:

1. Recorrer el raíz (`N`).
2. Recorrer el subárbol izquierdo (`I`).
3. Recorrer el subárbol derecho (`D`).

Dado las características recursivas de los árboles, el algoritmo de recorrido tiene naturaleza recursiva. Primero, se procesa el raíz; a continuación, el subárbol izquierdo, y, por último, el subárbol derecho. Para procesar el subárbol izquierdo, se hace una llamada recursiva al procedimiento *preorden*, y luego se hace lo mismo con el subárbol derecho.

El algoritmo recursivo correspondiente es:

```
si A no es vacío entonces
inicio
      ver los datos en el raíz de T
      Preorden (subárbol izquierdo del raíz de T)
      Preorden (subárbol derecho del raíz de T)
fin
```

Regla

En el recorrido preorden, el raíz se procesa antes que los subárboles izquierdo y derecho.

[3] El nombre *preorden* viene del prefijo latino *pre*, que significa «ir antes».

Si utilizamos el recorrido preorden del árbol de la Figura 20.22 se visita primero el raíz (nodo `A`). A continuación, se visita el subárbol `A`, que consta de los nodos `B,D` y `E`. Dado que el subárbol es a su vez un árbol, se visitan los nodos utilizando el orden `NID`. Por consiguiente, se visita primero el nodo `B`, después `D` (izquierdo) y, por último, `E` (derecho).

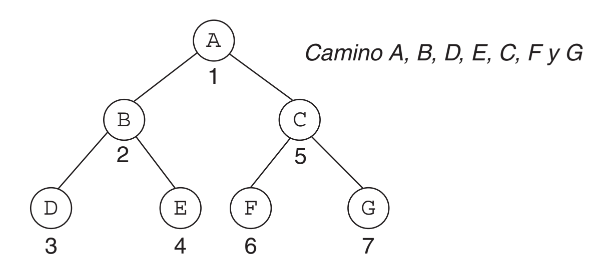


Figura 20.22. Recorrido preorden de un árbol binario.

A continuación, se visita subárbol derecho de `A`, que es un árbol que contiene los nodos `C, F` y `G`. De nuevo, siguiendo el orden `NID`, se visita primero el nodo `C`, a continuación `F` (izquierdo) y, por último, `G` (derecho). En consecuencia, el orden del recorrido preorden para el árbol de la Figura 20.22 es `A-B-D-E-C-F-G`. Un refinamiento del algoritmo es:

```
algoritmo preOrden  (val raíz <puntero nodos>)
Recorrer un árbol binario en secuencia nodo-izdo-dcho
Pre raiz es el nodo de entrada del árbol o subárbol
Post cada nodo se procesa en orden
1   si (raíz no es nulo)
    1   procesar (raíz)
    2   preOrden (raíz -> subarbolIzdo)
    3   preOrden (raíz -> subarbolDcho
    2   return
fin preorden
```

La función `PreOrden` muestra el código fuente en C++ del algoritmo ya citado anteriormente

```
void preorden (Nodo *p)
{
   if (p)
   {
      cout << p -> datos << ";
      PreOrden (p -> izda);
      PreOrden (p -> dcha);
   }
}
```

Gráficas de las llamadas recursivas de Preorden

El recorrido recursivo de un árbol se puede mostrar gráficamente por dos métodos distintos: 1) *paseo preorden del árbol*; 2) *recorrido algorítmico*.

Un medio gráfico para visualizar el recorrido de un árbol es imaginar que se está dando un «*paseo*» alrededor del árbol comenzando por la raíz y siguiendo el sentido contrario a las agujas del reloj, un nodo a continuación de otro sin pasar dos veces por el mismo nodo. El camino señalado por una línea continua

que comienza en el nodo A (Fig. 20.21) muestra el recorrido preorden completo. En el caso de la Figura 20.22 el recorrido es A-B-C-D-E-I.

El otro medio gráfico de mostrar el recorrido algorítmico recursivo es similar a las diferentes etapas del algoritmo. Así, la primera llamada procesa la raíz del árbol A. A continuación, se llama recursivamente a procesos del subárbol, procesa el nodo B. La tercera llamada procesa el nodo D, que es también subárbol D. En ese punto, se llama en *preorden*, con un puntero nulo, que produce un retorno inmediato al subárbol D que procesa a su subárbol derecho. Debido a que el subárbol derecho de D es también nulo, se vuelve al nodo B de modo que se puede procesar (visitar) su subárbol derecho, D. Después de procesar el nodo E, se hacen dos llamadas más, una con el puntero izquierdo *null* de E y otra con su puntero derecho *null*. Como el subárbol B ha sido totalmente procesado, se vuelve a la raíz del árbol y se procesa su subárbol derecho, E. Después de una llamada al subárbol izquierdo *null*, se llama al subárbol derecho. Aunque realizan todavía dos llamadas más, una al subárbol izquierdo *null* y otra al subárbol derecho. Entonces se retorna en el árbol, volviendo primero a C y, a continuación, a A, que concluye el recorrido del árbol.

20.6.2. Recorrido en orden

El recorrido ***en orden*** (*inorden*) procesa primero el subárbol izquierdo, después el raíz y, a continuación, el subárbol derecho. El significado de *in* es que la raíz se procesa entre los subárboles. Si el árbol no está vacío, el método implica los siguientes pasos:

1. Recorrer el subárbol izquierdo (I)
2. Visitar el nodo raíz (N)
3. Recorrer el subárbol derecho (D)

El algoritmo correspondiente es

```
Enorden(A)

si el árbol no esta vacío entonces
   inicio
      Recorrer el subárbol izquierdo
      Visitar el nodo raíz
      Recorrer el subárbol derecho
   fin
```

Un refinamiento del algoritmo es

```
algoritmo enOrden (val raíz <puntero a nodos>)
Recorrer un árbol binario en la secuencia izquierdo-nodo-derecho
pre raíz en el nodo de entrada de un árbol o subárbol
post cada nodo se ha de procesar en orden
1    si (raíz no es nulo)
     1 enOrden (raíz -> subárbolIzquierdo)
     2 procesar (raíz)
     3 enOrden (raíz->subárbolDerecho)
2    retorno
fin enOrden
```

En el árbol de la Figura 20.23, los nodos se han numerado en el orden en que son visitados durante el recorrido *enorden*. El primer subárbol recorrido es el subárbol izquierdo del nodo raíz (árbol cuyo

nodo contiene la letra `B`. Este subárbol consta de los nodos `B,D` y `E` y es a su vez otro árbol con el nodo `B` como raíz, por lo que siguiendo el orden `IND`, se visita primero `D`, a continuación `B` (nodo o raíz) y, por último, `E` (derecha). Después de la visita a este subárbol izquierdo se visita el nodo raíz `A` y, por último, se visita el subárbol derecho que consta de los nodos `D`, `F` y `G`. A continuación, siguiendo el orden `IND` para el subárbol derecho, se visita primero `F`, después `C` (nodo raíz) y, por último, `G`. Por consiguiente, el orden del recorrido `enOrden` de la Figura 20.23 es `D-B-E-A-F-C-G`.

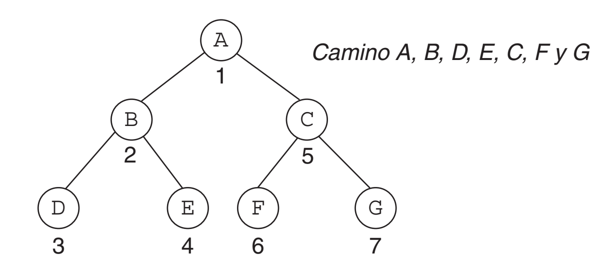


Figura 20.23. Recorrido en orden de un árbol binario.

La siguiente función visita y escribe el contenido de los nodos de un árbol binario de acuerdo al recorrido `EnOrden`. La función tiene como parámetro un puntero al nodo raíz del árbol.

```
void EnOrden (Nodo *p)
{
   if (p)
   {
      EnOrden(p -> izda);         // recorrer subárbol izquierdo
      cout << p -> datos << '';   // visitar la raíz
      EnOrden (p -> dcha);        // recorrer subárbol derecho
   }
}
```

20.6.3. Recorrido postorden

El recorrido *postorden* (`IDN`) procesa el nodo raíz (*post*) después de que los subárboles izquierdo y derecho se han procesado. Se comienza situándose en la hoja más a la izquierda y se procesa. A continuación, se procesa su subárbol derecho. Por último, se procesa el nodo raíz. Las etapas del algoritmo son:

1. `Recorrer el subárbol izquierdo (I).`
2. `Recorrer el subárbol derecho (D).`
3. `Recorrer el raíz (N).`

El algoritmo recursivo es

```
si A no está vacío entonces
   inicio
      Postorden (subárbol izquierdo del raíz de A)
      Postorden (subárbol derecho del raíz de A)
      Visualizar los datos del raíz de A
   fin
```

El refinamiento del algoritmo es

```
algoritmo postOrden (val raiz <puntero a nodod>)
   Recorrer un árbol binario en secuencia izquierda-derecha-nodo
   pre   raíz es el nodo de entrada de un árbol a un subárbol
   post   cada nodo ha sido procesado en orden

1 Si      (raíz no es nulo)
    1   postOrden (raíz -> SubarbolIzdo)
    2   postOrden    (raíz -> SubarbolDcho)
    3   procesar (raiz)
2 retorno
fin postOrden
```

Si se utiliza el recorrido *postorden* del árbol de la Figura 20.24, se visita primero el subárbol izquierdo A. Este subárbol consta de los nodos B, D y E y siguiendo el orden IDN, se visitará primero D (izquierdo), luego E (derecho) y por último B (nodo). A continuación, se visita el subárbol derecho de A que consta de los nodos C, F y G. Siguiendo el orden IDN para este árbol, se visita primero F (izquierdo), después G (derecho) y por último C (nodo). Finalmente, se visita el raíz A (nodo). Así, el orden del recorrido *postorden* del árbol de la Figura 20.24 es: D-E-B-F-G-C-A.

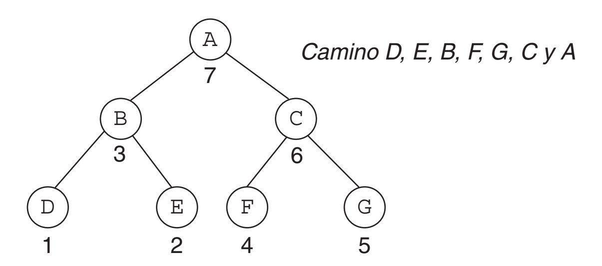


Figura 20.24. Recorrido *postorden* de un árbol binario.

La función PostOrden que implementa en C++ el código fuente del algoritmo correspondiente

```
void PostOrden (Nodo *p)
{
   if (p)
   {
      PostOrden (p -> izda);
      PostOrden (p -> dcha);
      cout << p -> datos << '';
   }
}
```

Notas de programación modular

La visita al nodo raíz del árbol que se representa mediante una sentencia cout podría representarse también con una función visitar

```
void Visitar (p)
{
   cout << p -> datos << '';
}
```

La función `EnOrden` quedaría así :

```
void EnOrden (Nodo *p)
{
   if (p)
   {
      EnOrden (p -> izda);
      Visitar (p);
      EnOrden (p -> dcha);
   }
}
```

Ejercicio 20.2

Implementar mediante plantillas (templates) las diferentes funciones recursivas del recorrido de un árbol binario.

Recorrido enOrden

```
// recorrido recursivo en orden
template <class T>
void enOrden (NodoArbol <T> *t, void visitar(T& item))
{

   //recorrido recursivo termina en un subarbol vacío
   if (t != NULL)
   {
      enOrden(t -> Izdo(), visitar);      // descenso
                                          // izdo
      visitar(t -> datos);                // visitar
                                          // el nodo
      enOrden(t -> Dcho(),visitar);       // descenso
                                          // dcho
   }
}
```

Recorrido postOrden

```
// recorrido recursivo PostOrden de los nodos
// en un árbol
template <class T>
void postOrden(NodoArbol<T> *t,void visitar(T& item))
{
   // el recorrido recursivo termina un subárbol vacío
      if (t != NULL)
      {
         postOrden (t->Izdo(), visitar);
         postOrden (t->Dcho(), visitar);
         visitar(t->datos);
      }
   }
```

Recorrido preOrden

```
// recorrido recursivo PreOrden
template <class T>
void preOrden (NodoArbol<T> *t, void visitar(T& item))
{
    if (t != NULL)
    {
      visitar  (t-> datos);
      preOrden (t-> Izdo(), visitar);
      preOrden (t-> Dcho(), visitar);
    }
}
```

Aplicación 20.1

Realizar mediante plantillas (templates) la especificación de un nodo y los recorridos de un árbol binario.

Clase nodo

```
template <class T>
class NodoArbolBin {
public :
   NodoArbolBin() {HijoIzdo = HijoDcho = 0;}
   NodoArbolBin(const T& e)
   {
      datos = e;
      HijoIzdo = HijoDcho = 0;
   }
   NodoArbolBin(const T& e, NodoArbolBin <T> *i, NodoArbolBin <T> *d)
   {
       datos = e;
       HijoIzdo = i;
       HijoDcho = d;
   }
private :
   T datos;
   NodoArbolBin <T> *HijoIzdo,      // subárbol izquierdo
                 *HijoDcho;          // subárbol derecho
};
```

Recorrido preorden

```
template <class T>
   void preOrden(NodoArbolBin<T> *t)
   { // recorrido preorden de *t
      if (t) {
         Visitar(t);                    // visitar raíz árbol
         preOrden(t -> HijoIzdo);      // subárbol izquierdo
         preOrden(t -> HijoDcho);      // subárbol derecho
   }
}
```

Recorrido en orden

```
template <class T>
   void enOrden(NodoArbolBin<T> *t)
   { // recorrido en orden de *t
      if (t) {
         enOrden(t -> HijoIzdo);
         Visitar(t);
         enOrden(t -> HijoDcho);
      }
   }
```

Recorrido postorden

```
template <class T>
   void postOrden(NodoArbolBin<T> *t)
   { // recorrido postorden de *t
      if (t) {
         postOrden(t ->HijoIzdo);     // subárbol izquierdo
         postOrden(t ->HijoDcho);     // subárbol derecho
         Visitar(t);                  // visitar árbol raíz

      }
   }
```

Si la función `Visitar()` se reemplaza por la sentencia

```
cont << t -> datos;
```

Ejercicio 20.3

Obtener los elementos de los tres recorridos fundamentales de los árboles siguientes.

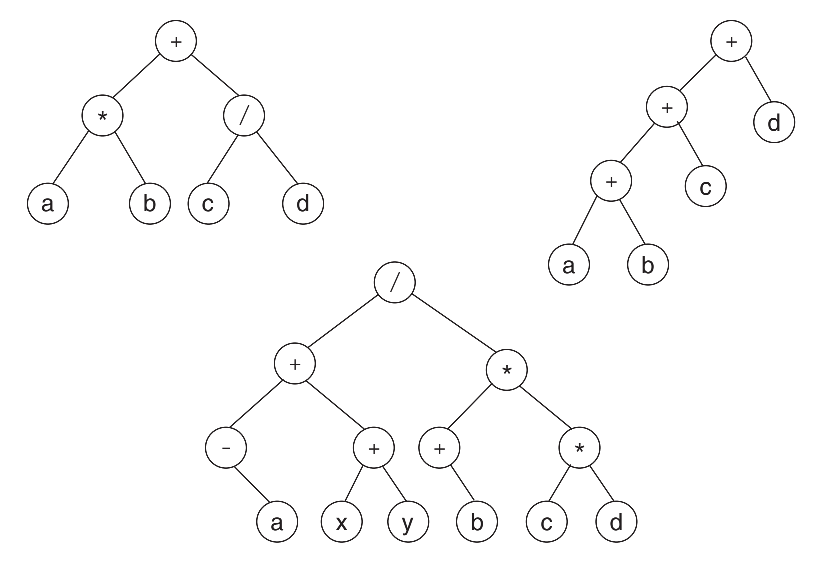


Figura 20.25. Árboles de expresión.

Los elementos de los árboles binarios listados en *preorden*, *enorden* y *postorden*.

	Árbol ***a***	*Árbol* ***b***	*Árbol* ***c***
PreOrden	`+*ab/cd`	`+++abcd`	`/*-a+xy*+ b* cd`
EnOrden	`a*b+c/d`	`a+b+c+d`	`-a+x+y/+ b*  c*d`
PostOrden	`ab*cd/+`	`ab+c+d+`	`a-xy++b+cd**/`

20.6.4. Profundidad de un árbol binario

La profundidad de un árbol binario es una característica que se necesita conocer con frecuencia durante el desarrollo de un programa. La función `Profundidad` evalúa la *profundidad* de un árbol binario. Para ello tiene un parámetro que es un puntero a la raíz del árbol.

El caso más sencillo de cálculo de la profundidad se produce cuando el árbol está vacío en cuyo caso la profundidad es 0. Si el árbol no está vacío, cada subárbol debe tener su propia profundidad, por lo que se necesita evaluar cada una por separado. Las variables `profundidadI`, `profundidadD` almacenarán las profundidades de los subárboles izquierdo y derecho respectivamente.

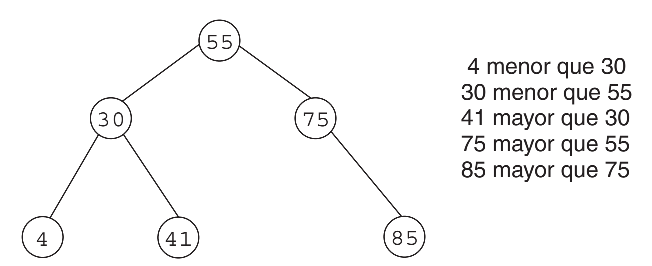
El método de cálculo de la profundidad de los subárboles utiliza llamadas recursivas a la función `Profundidad` con punteros a los respectivos subárboles como parámetros de la misma. La función `Profundidad` devuelve como resultado la profundidad del subárbol más profundo más 1 (la misma del raíz).

```
int Profundidad (Nodo *p)
{
   if (! p)
      return 0;
   else
   {
      int profundidadI = Profundidad (p -> izda);
      int profundidadD = Profundidad (p -> dcha);
      if (profundidadI > profundidadD)
         return profundidadI + 1;
      else
         return profundidadD + 1;
   }
}
```

20.7. ÁRBOL BINARIO DE BÚSQUEDA

Los árboles vistos hasta ahora no tienen un orden definido; sin embargo, los árboles binarios ordenados tienen sentido. Éstos se denominan árboles binarios de búsqueda, debido a que se puede buscar en ellos un término utilizando un algoritmo de búsqueda binaria similar al empleado en arrays.

Un **árbol binario de búsqueda** es aquel que dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que sus propios datos. El siguiente árbol sí es binario de búsqueda.



20.7.1. Creación de un árbol binario

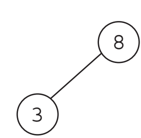
Supongamos que se desea almacenar los números

```
8    3    1    20    10    5    4
```

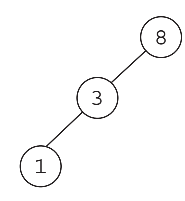
en un árbol binario de búsqueda. Siguiendo la regla, dado un nodo en el árbol todos los datos a su izquierda deben ser menores que todos los datos del nodo actual, mientras que todos los datos a la derecha deben ser mayores que los datos. Inicialmente el árbol está vacío y se desea insertar el `8`. La única elección es almacenar el `8` en el raíz:



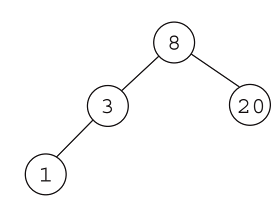
A continuación, viene el `3`. Ya que `3` es menor que `8`, el `3` debe ir en el subárbol izquierdo.



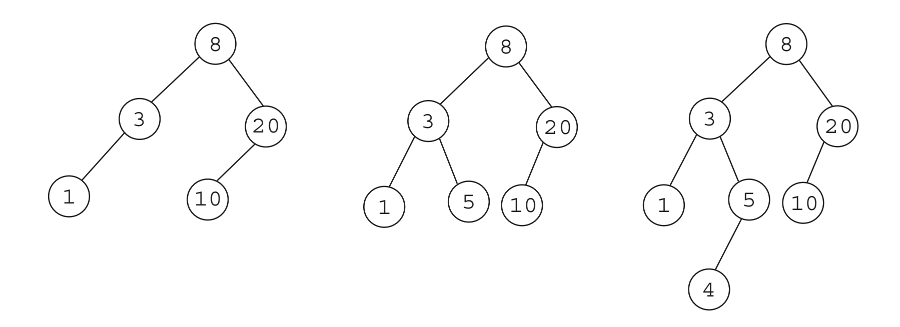
A continuación, se ha de insertar `1` que es menor que `8` y que `3`, por consiguiente irá a la izquierda y debajo de `3`.



El siguiente número es `20`, mayor que `8`, lo que implica debe ir a la derecha de `8`.



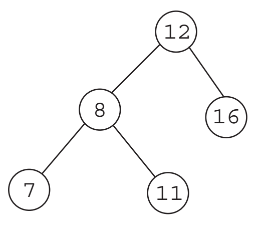
Cada nuevo elemento se inserta como una *hoja* del árbol. Los restantes elementos se pueden situar fácilmente.



Una propiedad de los árboles binarios de búsqueda es que no son únicos para los datos dados.

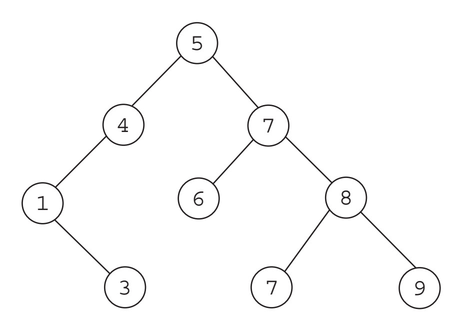
Ejemplo 20.9

Construir un árbol binario para almacenar los datos 12, 8, 7, 16 y 11.



Ejemplo 20.10

Construir un árbol binario de búsqueda que corresponda a un recorrido `enorden` *cuyos elementos son: 1, 3, 4, 5, 6, 7, 7, 8 y 9.*

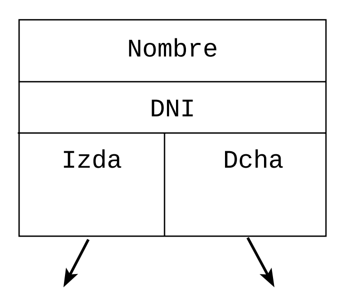


20.7.2. Implementación de un nodo de un árbol binario de búsqueda

Un árbol binario de búsqueda se puede utilizar cuando se necesita que la información se encuentre rápidamente. Estudiemos un ejemplo de árbol binario en el que cada nodo contiene información relativa a una persona. Cada nodo almacena un nombre de una persona y el número de matrícula en su universidad (dato entero).

Definición de la clase nodo

```
Nombre                    Tipo de dato, char (string)
DNI                       Tipo entero, int
```




```
class Nodo {
public :
   int nummat;
   char nombre[30];
   Nodo *izda, *dcha;
   Nodo (int id = 0, char *n = 0, Nodo *i = 0, Nodo *d = 0)
         : nummat(id), izda(i), dcha (d)
        { strcpy (nombre,n) ; }
};
```

20.8. OPERACIONES EN ÁRBOLES BINARIOS DE BÚSQUEDA

De lo expuesto se deduce que los árboles binarios tienen naturaleza recursiva y en consecuencia las operaciones sobre los árboles son recursivas, si bien siempre tenemos la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo.
- *Inserción* de un nodo.
- *Recorrido* de un árbol.
- *Borrado* de un nodo.

20.8.1. Búsqueda

La búsqueda de un nodo comienza en el nodo raíz y sigue estos pasos:

1. *La clave buscada se compara con la clave del nodo raíz.*
2. *Si las claves son iguales, la búsqueda se detiene.*
3. *Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.*

Buscar una información específica

Si se desea encontrar un nodo en el árbol que contenga la información sobre una persona específica. La función `buscar` tiene dos parámetros, un puntero al árbol y un número de matrícula para la persona requerida. Como resultado, la función devuelve un puntero al nodo en el que se almacena la información

sobre esa persona; en el caso de que la información sobre la persona no se encuentra, se devuelve el valor 0. El algoritmo de búsqueda es el siguiente:

1. *Comprobar si el árbol está vacío, en caso afirmativo, se devuelve 0. Si la raíz contiene la persona, la tarea es fácil: el resultado es, simplemente, un puntero a la raíz.*
2. *Si el árbol no está vacío, el subárbol específico depende de que el número de matrícula requerido es más pequeño o mayor que el número de matrícula del nodo raíz.*
3. *La función de búsqueda se consigue llamando recursivamente a la función buscar con un puntero al subárbol izquierdo o derecho como parámetro.*

El código C++ de la función buscar es:

```
Nodo *buscar (Nodo *p, int buscado)
{
   if (! p)
      return 0;
   else if (buscado == p -> nummat)
      return p;
   else if (buscado < p -> nummat)
      return buscar (p -> izda, buscado);
   else
      return buscar (p -> dcha, buscado);
}
```

20.8.2. Insertar un nodo

Una característica fundamental que debe poseer el algoritmo de inserción es que el árbol resultante de una inserción en un árbol de búsqueda ha de ser también de búsqueda. En esencia, el algoritmo de inserción se apoya en la localización de un elemento, de modo que si se encuentra el elemento (*clave*) buscado, no es necesario hacer nada; en caso contrario, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en el caso de existir).

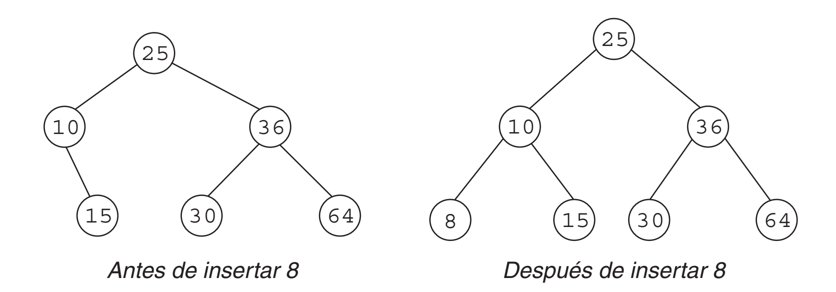
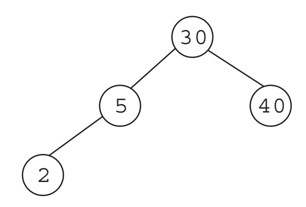


Figura 20.26. Inserción en un árbol binario de búsqueda.

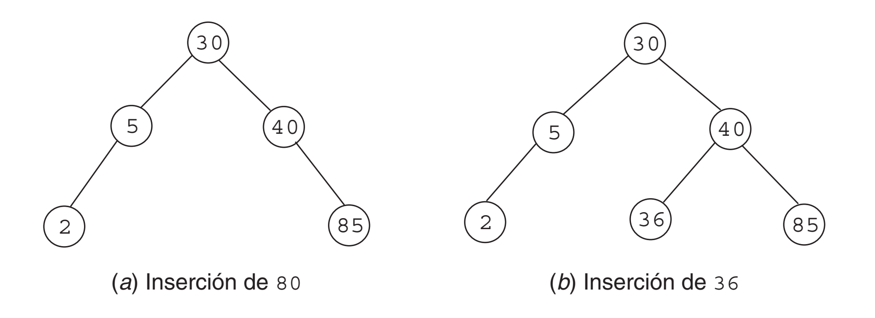
Por ejemplo, considérese el caso de añadir el nodo `8` al árbol de la Figura 20.26. Se comienza el recorrido en el nodo raíz `25`; la posición 8 debe estar en el subárbol izquierdo de `25(8 < 25)`. En el nodo `10`, la posición de `8` debe estar en el subárbol izquierdo de `10`, que está actualmente vacío. El nodo `8` se introduce como un hijo izquierdo del nodo `10`.

Ejemplo 20.11

Insertar un elemento con clave 80 en el árbol binario de búsqueda siguiente:



A continuación, insertar un elemento con clave 36 en el árbol binario de búsqueda resultante.



(*a*) Inserción de `80` (*b*) Inserción de `36`

20.8.3. Insertar nuevos nodos

La función `insertar` que pone nuevos nodos es sencilla. Se deben declarar tres argumentos: un puntero al raíz del árbol y al nuevo nombre y número de matrícula de la persona. El procedimiento creará un nuevo nodo para la nueva persona y lo inserta en el lugar correcto en el árbol de modo que el árbol permanezca como binario de búsqueda.

```
void insertar (Nodo * p, int nuevo_mat, char  *nuevo_nombre)
{
   if (! p)
      p = new Nodo (nuevo_mat, nuevo_nombre);
   else if (nuevo_mat < p -> nummat);
      insertar (p -> izda, nuevo_mat, nuevo_nombre);
   else
      insertar (p -> dcha, nuevo_mat, nuevo_nombre);
}
```

Si el árbol está vacío, es fácil insertar la entrada en el lugar correcto. El nuevo nodo es la raíz del árbol y el puntero `p` se pone apuntando a ese nodo. El parámetro `p` debe ser un parámetro referencia ya que debe ser leído y actualizado. Si el árbol no está vacío, se debe elegir entre insertar el nuevo nodo en el subárbol izquierdo o derecho, dependiendo de que el número de matrícula de la nueva persona sea más pequeño o mayor que el número de matrícula en la raíz del árbol. La operación de *inserción* de un nodo es una extensión de la operación de búsqueda. Los pasos a seguir son:

1. *Asignar memoria para una nueva estructura nodo.*
2. *Buscar en el árbol para encontrar la posición de inserción del nuevo nodo, que se colocará como nodo hoja.*
3. Enlazar el nuevo nodo al árbol.

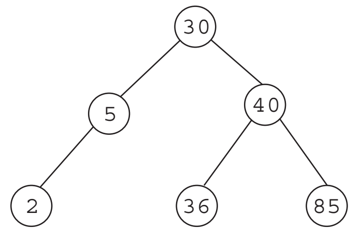
20.8.4. Eliminación

La operación de *eliminación* de un nodo es también una extensión de la operación de búsqueda, si bien es más compleja que la inserción debido a que el nodo a suprimir puede ser cualquiera y la operación de supresión debe mantener la estructura de árbol binario de búsqueda después de la eliminación de datos. Los pasos a seguir son:

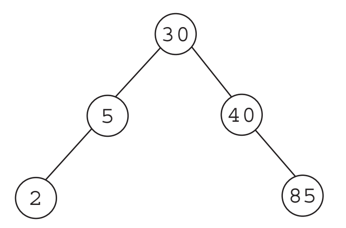
1. *Buscar en el árbol para encontrar la posición de nodo a eliminar.*
2. *Reajustar los punteros de sus antecesores si el nodo a suprimir tiene menos de 2 hijos, o subir a la posición que éste ocupa el nodo más próximo en clave (inmediatamente superior o inmediatamente inferior) con objeto de mantener la estructura de árbol binario.*

Ejemplo 20.12

Suprimir el elemento de clave 36 del siguiente árbol binario de búsqueda:

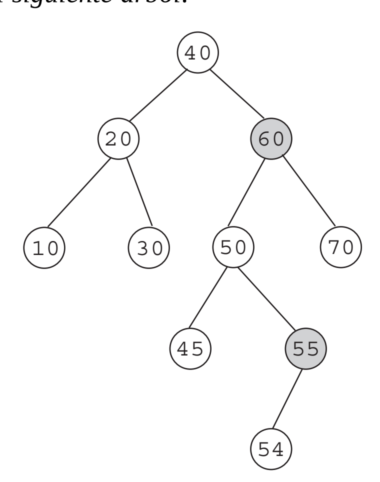


El árbol resultante es

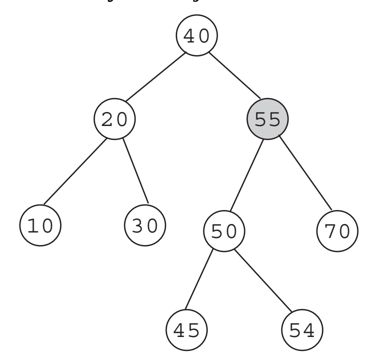


Ejemplo 20.13

Borrar el elemento de clave 60 del siguiente árbol:



Se reemplaza 60 bien con el elemento mayor (55) en su subárbol izquierdo o el elemento más pequeño (70) en su subárbol derecho. Si se opta por reemplazar con el elemento mayor del subárbol izquierdo, se mueve el 55 al raíz del subárbol y se reajusta el árbol.



20.8.5. Recorrido de un árbol

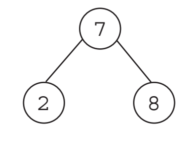
Existen dos tipos de recorrido de los nodos de un árbol: el recorrido en anchura y el recorrido en profundidad. En el *recorrido en anchura* se visitan los nodos por niveles. Para ello se utiliza una estructura auxiliar tipo cola en la que después de mostrar el contenido de un nodo, empezando por el nodo raíz, se almacenan los punteros correspondientes a sus hijos izquierdo y derecho. De esta forma si recorremos los nodos de un nivel, mientras mostramos su contenido, almacenamos en la cola los punteros a todos los nodos del nivel siguiente.

El *recorrido en profundidad* se realiza por uno de tres métodos recursivos: *preorden, inorden* y *postorden*. El primer método consiste en visitar el nodo raíz, su árbol izquierdo y su árbol derecho, por este orden. El recorrido *inorden* visita el árbol izquierdo, a continuación el nodo raíz y finalmente el árbol derecho. El recorrido *postorden* consiste en visitar primero el árbol izquierdo, a continuación el derecho y finalmente el raíz.

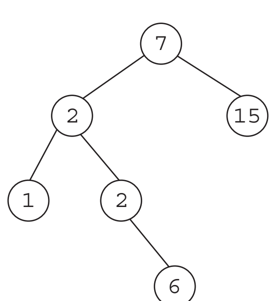
Preorden	Raíz	Izdo	Dcho
En orden	Izdo	Raíz	Dcho
Postorden	Izdo	Dcho	Raíz

20.8.6. Determinación de la altura de un árbol

La *altura* de un árbol dependerá del criterio que se siga para definir dicho concepto. Así, si en el caso de un árbol que tiene nodo raíz se considera que su altura es 1, la altura del árbol



es 2, y la altura del árbol



es 4. Por último, si la altura de un árbol con un nodo es 1, la altura de un árbol vacío (el puntero es *nul*) es 0.

> **Nota**
>
> La altura de un árbol es 1 más que la mayor de las alturas de sus subárboles izquierdo y derecho.

20.9. APLICACIONES DE ÁRBOLES EN ALGORITMOS DE EXPLORACIÓN

Los algoritmos recursivos de recorridos de árboles son el fundamento de muchas aplicaciones de árboles. Proporcionan un acceso ordenado y metódico a los nodos y a sus datos. Vamos a considerar en esta sección una serie de algoritmos de recorrido usuales en numerosos problemas de programación, tales como contar el número de nodos hoja, calcular la profundidad de un árbol, imprimir un árbol o copiar y eliminar árboles.

20.9.1. Visita a los nodos de un árbol

En muchas aplicaciones se desea explorar (recorrer) los nodos de un árbol pero sin tener en cuenta un orden de recorrido preestablecido. En esos casos, el cliente o usuario es libre para utilizar el algoritmo oportuno.

La función `ContarHojas` recorre el árbol y cuenta el número de nodos hijo. Para realizar esta operación se ha de visitar cada nodo comprobando si es un nodo hoja. El recorrido utilizado será el *postorden*.

```
// Función ContarHojas
// la función utiliza recorrido postorden
// en cada visita se comprueba si el nodo es un nodo hoja
// (no tiene descendientes)

template << class T>
void ContarHojas (NodoArbol <T> *t, int& cuenta)
{
   // utilizar descenso postorden
   if (t != NULL)
   {
      ContarHoja (t -> Izdo(), cuenta);      // descenso izquierdo
      ContarHoja (t -> Dcho(), cuenta);      // descenso derecho

      // verificar si nodo t es un nodo hoja
      // en caso afirmativo, incrementar la variable cuenta
      if (t -> Izdo() == NULL && t -> Dcho() == NULL)
         cuenta ++;
}
```

La función `Profundidad` utiliza un recorrido postorden para calcular la profundidad de un árbol binario. En cada nodo se calcula la profundidad de los subárboles izquierdo y derecho. La profundidad resultante del nodo es 1 más que la profundidad máxima de sus subárboles.

```
// Función Profundidad
// Recorrido postorden
// Calcula la profundidad de los subárboles izquierdo y derecho
// de un nodo y devuelve la profundidad como 1 + max (profIzda,
// profDcha)
// La profundidad de un árbol vacío es 0

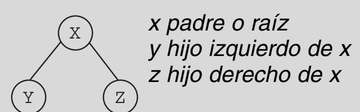
template <class T >
void Profundidad (NodoArbol <T> *t)
{
   int profIzdo, profDcho, valorprof;

   if (t == NULL)
      valorprof = 0;
   else
   {
      profIzdo = Profundidad (t —>Izdo());
      profDcho = Profundidad (t —> Dcho());
      valorprof = 1 + (profIzdo > profDcho ? profIzdo :
                        profDcho);
   }
   return valorprof ;
}
```

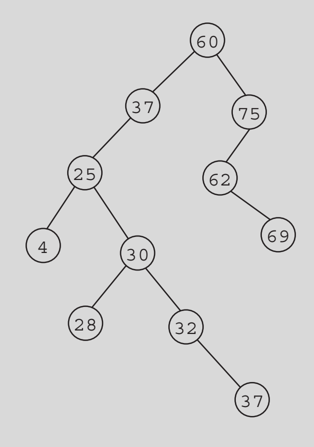
RESUMEN

En este capítulo se introdujo y desarrolló la estructura de datos dinámica árbol. Esta estructura, muy potente, se puede utilizar en una gran variedad de aplicaciones de programación.

La estructura árbol más utilizada normalmente es el *árbol binario*. Un **árbol binario** es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho.



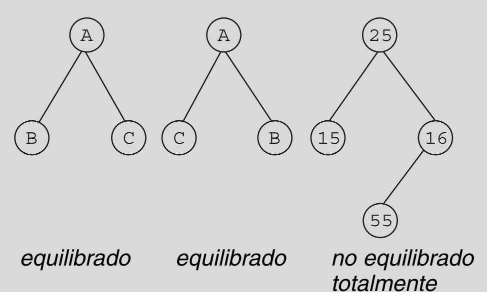
En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero sí cada elemento restante tiene un padre. `X` es un *antecesor* o *ascendente* del elemento `Y`.



La altura de un árbol binario es el número de ramas entre el raíz y la hoja más lejana, más 1. Si el árbol A

es vacío, la altura es 0. La altura del árbol anterior es 6. El *nivel* o *profundidad* de un elemento es un concepto similar al de altura. En el árbol anterior el nivel de 30 es 4 y el nivel de 37 es 6. Un nivel de un elemento se conoce también como *profundidad*.

Un árbol binario no vacío está *equilibrado totalmente* si sus subárboles izquierdo y derecho tienen la misma altura y ambos son o bien vacíos o totalmente equilibrados.



equilibrado *equilibrado* *no equilibrado totalmente*

Los árboles binarios presentan dos tipos característicos: *árboles binarios de búsqueda* y *árboles binarios de expresiones*. Los árboles binarios de búsqueda se utilizan fundamentalmente para mantener una colección ordenada de datos y los árboles binarios de expresiones para almacenar expresiones.

EJERCICIOS

20.1. Escribir un programa que procese un árbol binario cuyos nodos contengan caracteres y a partir del siguiente menú de opciones:

`I` (seguido de un carácter)	: Insertar un carácter
`B` (seguido de un carácter)	: Buscar un carácter
`RE`	: Recorrido en orden
`RP`	: Recorrido en preorden
`RT`	: Recorrido postorden
`SA`	: Salir

20.2. Escribir una función que tome un árbol como entrada y devuelva el número de hijos del árbol.

20.3. Escribir una función *booleana* a la que se le pase un puntero a un árbol binario y devuelva verdadero (*true*) si el árbol es completo y false en caso contrario.

20.4. Diseñar una función recursiva que devuelva un puntero a un elemento en un árbol binario de búsqueda.

20.5. Diseñar una función iterativa que encuentre un elemento en un árbol binario de búsqueda.

PROBLEMAS

20.1. Crear un archivo de datos en el que cada línea contenga la siguiente información:

Columnas	1-20	Nombre
	21-31	Número de la Seguridad Social
	32-78	Dirección

Escribir un programa que lea cada registro de datos de un árbol, de modo que cuando el árbol se recorra utilizando recorrido en orden, los números de la seguridad social se ordenen en orden ascendente. Imprimir una cabecera «DATOS DE EMPLEADOS ORDENADOS POR NÚMERO SEGURIDAD SOCIAL». A continuación, se han de imprimir los tres datos utilizando el siguiente formato de salida:

Columnas	1-11	Número de la Seguridad Social
	25-44	Nombre
	58-104	Dirección

20.2. Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto, así como su frecuencia de aparición.

20.3. Se dispone de un árbol binario de elementos de tipo int. Escribir funciones que calculen:

a) La suma de sus elementos.
b) La suma de sus elementos que son múltiplos de 3.

20.4. Escribir una función booleana IDÉNTICOS que permita decir si dos árboles binarios son iguales.

20.5. Diseñar un programa interactivo que permita dar altas, bajas, listar, etc., en un árbol binario de búsqueda.

20.6. Diseñar procedimientos no recursivos que listen los nodos de un árbol en inorden, preorden y postorden.

20.7. Dados dos árboles binarios de búsqueda, indicar mediante un programa si los árboles tienen o no elementos comunes.

20.8. Dado un árbol binario de búsqueda construir su árbol espejo. (Árbol espejo es el que se construye a partir de uno dado, convirtiendo el subárbol izquierdo en subárbol derecho y viceversa.)

20.9. Un árbol binario de búsqueda puede implementarse con un array. La representación no enlazada correspondiente consiste en que para cualquier nodo del árbol almacenado en la posición **I** del array, su hijo izquierdo se encuentra en la posición **2*I** y su hijo derecho en la posición **2*I + 1**. Diseñar a partir de esta representación las funciones para gestionar interactivamente un árbol de números enteros. (Comente el inconveniente de esta representación de cara al máximo y mínimo número de nodos que pueden almacenarse.)

20.10. Una matriz de *N* elementos almacena cadenas de caracteres. Utilizando un árbol binario de búsqueda como estructura auxiliar, ordene ascendentemente la cadena de caracteres.

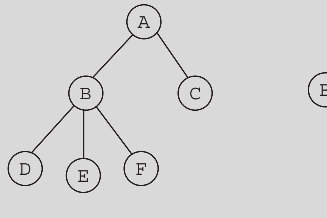
20.11. Dado un árbol binario de búsqueda, diseñe un procedimiento que liste los nodos del árbol ordenados descendentemente.

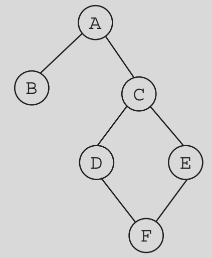
EJERCICIOS RESUELTOS

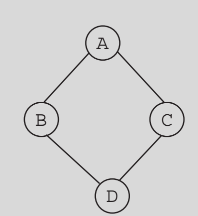
1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 390).

2. Sitio web del libro, *www.mhe.es/joyanes* (código fuente).

20.1. *Explicar porqué cada una de las siguientes estructuras no es un árbol binario.*



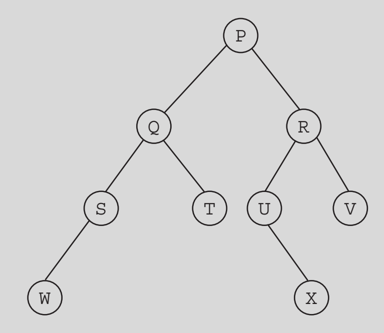




20.2. *Considérese el árbol siguiente.*

a) *¿Cuál es su altura?*
b) *¿Está el árbol equilibrado? ¿Por qué?*

c) *Listar todos los nodos hoja.*
d) *¿Cuál es el predecesor inmediato (padre) del nodo U?*
e) *Listar los hijos del nodo R.*
f) *Listar los sucesores del nodo R.*



20.3. *Para el árbol del ejercicio anterior realizar los siguientes recorridos: RDI, DRI, DIR.*

20.4. *Para cada una de las siguientes listas de letras:*

a) *Dibujar el árbol binario de búsqueda que se construye cuando las letras se insertan en el orden dado.*

b) *Realizar recorridos inorden, preorden y postorden del árbol y mostrar la secuencia de letras que resultan en cada caso.*

(I) `M,Y,T,E,R`
(II) `R,E,M,Y,T`
(III) `T,Y,M,E,R`
(IV) `C,O,R,N,F,L,A,K,E,S`

20.5. *Dibujar los árboles binarios que representan las siguientes expresiones:*

a) `(A + B) / (C – D)`
b) `A + B + C / D`
c) `A-(B -(C - D) / (E + F))`
d) `(A + B) * ((C + D) / (E + F))`
e) `(A - B) / ((C * D) - (E / F))`

20.6. *El recorrido preorden de un cierto árbol binario produce `ADFGHKLPQRWZ`, y en recorrido inorden produce `GFHKDLAWRQPZ`. Dibujar el árbol binario.*

PROBLEMAS RESUELTOS

1. Schaum (McGraw-Hill) de Joyanes, L. y Sánchez, L. *Programación en C++* (análisis y código fuente, pág. 392).

2. Sitio web del libro, *`www.mhe.es/joyanes`* (código fuente).

20.1. *Codifique la clase* `arbol` *como clase amiga de una clase* `Nodo` *y que permita realizar métodos recursivos. La clase árbol debe contener las funciones miembro* `Obtener` *y poner tanto el hijo izquierdo y derecho como el elemento de la raíz del árbol.*

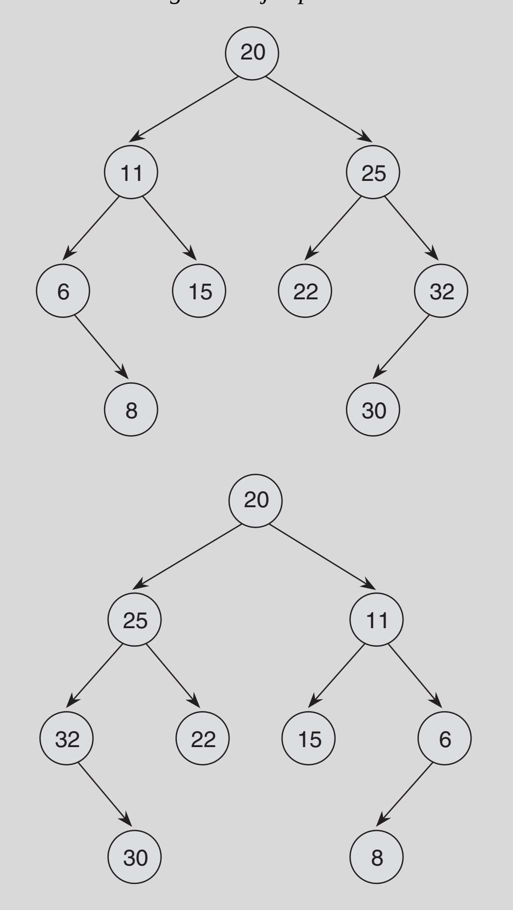
20.2. *Añadir a la clase* `arbol` *del ejercicio anterior el constructor de copia.*

20.3. *Añadir a la clase árbol del Problema 20.1 una función miembro que copie el objeto actual en un objeto que reciba como parámetro.*

20.4. *Sobrecargue el operador de asignación de la clase* `arbol` *para permitir asignar árboles completos.*

20.5. *Escriba un método de la clase* `arbol` *que realice el espejo del objeto árbol. El espejo de un*

árbol es otro árbol que es el mismo que el que se ve si se reflejara en un espejo, tal y como se indica en el siguiente ejemplo:



20.6. *Escribir funciones miembro de la clase árbol del Problema 20.1 para hacer los recorridos recursivos en profundidad inorden, preorden, postorden.*

20.7. *Escribir una función miembro de la clase* `arbol` *que nos cuente el número de nodos que tiene.*

20.8. *Añadir a la clase* `arbol` *un método que calcule el número de hojas del objeto.*

20.9. *Añadir a la clase árbol una función miembro que reciba como parámetro un número natural* `n` *que indique un nivel, y muestro todos los nodos del árbol que se encuentren en ese nivel.*

20.10. *Construir una función miembro de la clase* `arbol` *para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado.*

20.11. *Escribir una función miembro de la clase* `arbol` *recursiva y otra iterativa que se encargue de insertar la información dada en un elemento en un* `arbol` *binario de búsqueda.*

20.12. *Escribir funciones miembros de la clase* `arbol` *iterativas y recursivas para borrar un elemento de un árbol binario de búsqueda.*

REFERENCIAS BIBLIOGRÁFICAS

Aho, Alfred V., y Ullman, Jeffrey D., *Foundations of Computer Science*, Computer Science Press, 1992.

Cormen, Thomas H.; Leiserson Charles E., y Rivert Ronal, L., *Introduction to Algorithms*, The Mit Press, McGraw-Hill, 1992.

Carrasco, Hellman y Veroff, *Data Structures and problem solving with Turbo Pascal*, The Benjamin/Cummings, 1993.

Ceri, Stefano; Mandrioli, Dino, y Sbattella, Licia, *The Art & Craft of Computing*. Harlow, England, Addison-Wesley, 1997.

Collins, William J., *Data structures. An Object-Oriented Approach*, Addison-Wesley, 1992.

Franch Gutierrez, Xavier, *Estructura de datos. Especificación, Diseño e implementación*, Barcelona. Edicions UPC, 1994.

Gilberg, Richard F., y Forouzan, Behrouz A., *Data Structures*, Boston, ITP, 1998.

Hale, Guy J., y Easton, Richard J., *Applied Data Structures Using Pascal*, Heath, Massachusetts, 1987.

Horowitz, Ellis, y Sartaj, Sahni, *Data Structures in Pascal*, 3.ª edición, Nueva York, Computer Science Press, 1990.

Horowitz, Ellis, y Sartaj, Sahni, *Fundamentals of data Structures in C*, Nueva York, Computer Science Press, 1993.

Joyanes, Luis, *Fundamentos de programación*, 2.ª edición. Madrid, McGraw-Hill, 1996.

Joyanes, Luis, y Zahonero, Ignacio, *Estuctura de datos*, Madrid, McGraw-Hill, 1998.

Joyanes, L.; Zahonero, I., y Hermoso, A., *Pascal y Turbo Pascal. Un enfoque práctico*. Madrid, McGraw-Hill, 1995.

Joyanes, L.; Zahonero, I.; Sánchez, L.; Fernández, M., y Centenera, P., *Estructura de datos. Libro de problemas*. Madrid, McGraw-Hill, 1999.

Kruse, Robert L., *Data Structures and program design*, Prentice-Hall, 1994.

Koffman, Elliot B., y Maxim, Bruce R., *Software Design and Data Structures in Turbo Pascal*, Addison-Wesley, 1994.

Salmon, William J., *Structures and abstractions*, Irwin, 1991.

Sahni, Sartaj., *Data Structures, Algorithms, and Applications in C++*, Boston, McGraw-Hill, 1998.

Tenembaum Aaron M., y Angenstein Moshe, *Data structures using Pascal*, Prentice-Hall, 1986.

Wirth, *Algoritmos y Estructuras de datos*, Madrid, Ed. del Castillo, 1985.

P A R T E IV

PROGRAMACIÓN AVANZADA EN C++

CAPÍTULO 21
Sobrecarga de operadores

Contenido

INTRODUCCIÓN

La sobrecarga de operadores es una de las características más «fascinantes» de C++ y, naturalmente, de la programación orientada a objetos. La ***sobrecarga de operadores*** hace posible manipular objetos de clases con operadores estándar tales como `+`, `*`, `[ ]` y `<<`. Esta propiedad de los operadores permite redefinir el lenguaje C++, que puede crear nuevas definiciones de operadores. En este capítulo se mostrará la sobrecarga de operadores unitarios y binarios, incluyendo el operador de moldeado (***cast***) para conversión de tipos implícitos y explícitos, operadores relacionales, operador de asignación y otros operadores.

CONCEPTOS CLAVE

- Conversión de tipos de datos.
- Función operador.
- Moldeado (*cast*).
- Operador binario.
- Operador unitario.
- `Operator`.
- Sobrecarga de `new` y `delete`.

21.1. SOBRECARGA

En C++ hay un número determinado de operadores predefinidos que se pueden aplicar a tipos estándar incorporados o integrados. Por ejemplo, el operador + sirve para sumar dos variables de tipo `int`, el operador <= puede comparar dos valores de tipo `double`. Estos operadores predefinidos, realmente, están sobrecargados ya que cada uno de ellos se puede utilizar para diferentes tipos de datos. Por ejemplo, un operador + se utiliza para tipo `int` y otro para tipo `double`. Sin embargo, cuando los tipos de datos son clases no hay prácticamente operadores predefinidos.

Al trabajar con clases; por ejemplo, supóngase una clase `Racional` que representa a los números racionales y sus posibles operaciones, es muy importante que se pudieran realizar operaciones tales como suma, resta, producto, etc., de igual forma que se realizan las operaciones de números enteros, reales, etc. Es decir, si `r1` y `r2` son números de tipo racional o de tipo complejo, se trata de definir que el operador + pueda servir para escribir `r1+r2` y que el compilador entienda correctamente la operación. En esencia lo que se busca es que los tipos de dato, tales como clases (`String, Fecha, Complejo`, etc.), puedan ser tratados como si fueran tipos de datos simples.

Casi todos los operadores predefinidos para tipos estándar se pueden sobrecargar. La Tabla 21.1 muestra dichos operadores.

Tabla 21.1. Operadores que se pueden sobrecargar

`new`	`delete`	`new[]`	`delete[]`					
`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`	`~`
`!`	`=`	`<`	`>`	`+=`	`-=`	`*=`	`/=`	`%=`
`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`	`<<=`	`==`	`!=`
`<=`	`>=`	`&&`	`||`	`++`	`--`	`,`	`->*`	`->`
`()`	`[]`							

Un *operador unitario* es un operador que tiene un único operando. El operador ++, por ejemplo, es unitario. Un *operador binario*, por el contrario, tiene dos operandos. El operador /, por ejemplo, es binario. Algunos operadores tales como + son unitarios y binarios, dependiendo de su uso.

```
N++;     // unitario
x / y;     // binario
+x;      // unitario
x + y;   // binario
```

Una función operador es una función cuyo nombre consta de la palabra reservada `operator` seguida por un operador unitario o binario.

Formato

```
operator operador
```

Los siguientes operadores no se pueden sobrecargar:

`.`	acceso a miembro
`.*`	indirección de acceso a miembro
`::`	resolución de ámbitos
`?:`	`if` aritmético

Excepto para el caso de los operadores `new, delete` y `->`, las funciones operador pueden devolver cualquier tipo de dato. La precedencia, agrupamiento y número de operandos no se puede cambiar. Con excepción del operador `()`, las funciones operador no pueden tener argumentos por defecto.

Una función operador debe ser o bien una función miembro no estática o una función no miembro que tenga al menos un parámetro cuyo tipo es una clase, referencia o una clase enumeración, o referencia a una enumeración. Esta regla evita cambiar el significado de operadores que operan sobre tipos de datos intrínsecos. Por ejemplo, el siguiente código no será válido.

```
int operator +(int x, int y)
{
   return x * y;
}
```

Por el contrario, la siguiente función no miembro es válida ya que al menos un parámetro es un tipo clase:

```
class Integer {
// ...
public:
   int valor;
};
int operator + (const Integer &x, int y)
{
   return x.valor + y;
}
```

Cuando se declaran nuevos y propios operadores para clases lo que se hace es escribir una función con el nombre `operatorX` en donde `x` es el símbolo de un operador. Así, si se escribe `operator+`, la función que se declara tendrá el nombre `operator+`. Existen dos medios de construir nuestros propios operadores, bien formulándolos como funciones amigas, aunque lo usual será como funciones miembro.

Reglas

- Las funciones con el nombre **`operatorX`**, en donde X es el símbolo del operador.
- La mayoría de los operadores predefinidos pueden ser sobrecargados para tipos clase.
- Puede tener un operando (unitario) o dos operandos (binario).
- El número de operandos y la prioridad son la misma que la del correspondiente operador predefinido.
- Se puede construir como una función miembro o como una función amiga.

21.2. OPERADORES UNITARIOS

Los *operadores unitarios* son:

`new`	`delete`	`new[] delete`
`++`	incremento	
`--`	decremento	
`()`	llamada a función	
`[]`	indexación	

+	más
-	menos
*	desreferencia (indirección)
&	dirección de
!	NOT lógico
~	NOT bit a bit
,	coma

El principio de la sobrecarga tiene gran aplicación en operadores *unitarios*.

Ejemplo 21.1

Definición del signo menos para definir vectores opuestos.

```
vector u,v;
...
v = -u;
```

Una vez que se han definido el operador unitario menos y un operador binario más, se podría definir un operador binario menos de la siguiente forma:

```
a - b = a + (-b).
```

Ejemplo 21.2

```
//operador unitario junto con operadores binarios
#include <iostream.h>
class vector {
public:
   vector(float xx = 0, float yy = 0)-:x(xx), y(yy) {}
   void imprimirvec() const;
   vector operator+(const vector &b)const; // más binario
   vector operator -();                    // menos unitario
   vector operator -(const vector &b)const;// menos binario
private:
   float x, y;
};

   void vector::imprimirvec() const
{
   cout << x << ´´ << y << endl;
}

vector vector::operator+ (const vector &b)const // más binario
{
   return vector(x + b.x, y + b.y);
}

vector vector::operator-()              // menos unitario
{
   return vector(-x,-y);
}
```

```
vector vector::operator-(const vector &b)const // menos binario
{
return *this + -b; // *this es el objeto apuntado por this
}
int main()
{
      vector u(3,1), v(1,2), suma, neg, difer;
      suma = u + v ;
      suma.imprimirvec();    // salida (4,3)
      neg = -suma;
      neg.imprimirvec();     // salida (-4, -3)
      difer = u-v ;
      difer.imprimirvec();   // salida (2, -1)
      return 0;
}
```

21.3 SOBRECARGA DE OPERADORES UNITARIOS

Los operadores unitarios son aquellos que actúan sobre un único operando. Ejemplos de operadores unitarios son:

```
++    --    -
```

Consideremos una clase de tipo `t` y un objeto `x` de tipo `t`. Se define un operador unitario sobrecargado ++, y

```
++x
```

se interpreta como

```
operator++(x)
```

o bien como

```
x.operator++()
```

Un operador unitario *se puede definir bien como* un amigo con un argumento, *o bien como* una función miembro sin argumento, que es el caso más frecuente, *pero nunca* los dos a la vez.

Para sobrecargar el operador ++, por ejemplo, se utiliza la declaración

```
void operator ++
```

Esta sintaxis indica al compilador que llame a esta función miembro siempre que se encuentre el operador ++.

El operador ++ se utiliza normalmente para incrementar un objeto de una clase `tiempo`. Examinaremos, en primer lugar, cómo mediante una función miembro (método) se incrementaría el tiempo.

```
void incTiempo         // añadir 1 segundo al tiempo
{
   seg++ ;
   if(segs > 59) {segs -= 60 ; ++mins;}
   if(mins > 59) {mins -= 60 ; ++hrs;}
}
```

Para incrementar un tiempo `t1` se llamaría al método en la forma usual:

```
t1.incTiempo();           // incrementar t1
```

En el siguiente ejemplo utilizamos el mismo método, pero ahora se invoca utilizando el operador ++ en lugar del nombre incTiempo.

```
// archivo TIEMPO.CPP
// utilizar operador sobrecargado ++ para incrementar objetos de la
//clase tiempo

#include <iostream>
#include <stdio.h>
using namespace std;

class tiempo
{
   private:
      int hrs;        // horas
      int mins;       // minutos
      int segs;       // segundos

      public.
         tiempo()
            {segs = 0;mins = 0;hrs = 0;} // inicializa tiempo a 00:00:00

         void verHora(void)        // visualizar tiempo
            {cout << hrs << ":" << mins << ":" << segs;}

         void leerTiempo (void)    // leer hora del teclado
            {cin >> hrs >> min >> segs;}

         tiempo & void operator ++ // sumar 1 segundo a esta hora
            {
            segs++;
            if (segs > 59) {segs -= 60; ++mins;}
            if (mins > 59) {mins -= 60; ++hrs;}
            return * this;
         };

   int main()
   {
      tiempo t1;           // declara variable tiempo
         cout << "\n Introduzca hora (hh :mm-:ss) : ";
      t1.leerTiempo();     // leer t1

      cout <<  "visualizar t1";
      t1.verHora();

      ++t1;                // incrementar t1,

      cout << "\n después de incrementar, t1= "; // ver de nuevo t1
      t1.verHora();
   }
```

En lugar de utilizar el nombre de la función `incTiempo` para definir el método, utilizamos el operador sobrecargado ++

```
tiempo & operator++
```

Se puede entonces incrementar `t1` un objeto de la clase `tiempo` con la sentencia

```
++t1;
```

Un ejemplo de salida de esta programa puede ser:

```
Introduzca hora (hh:mm:ss):10:59:59
Después de incrementar, t1 = 11:00:00
```

Otro ejemplo de un operador unitario sobrecargado

Consideremos un tipo de clase `vector` que contiene:

- Dos datos miembro privados (coordenadas de tipo double).
- Dos funciones `public`.
  - — Una función miembro de inicialización.
  - — Una función miembro que visualiza coordenadas.
- Un operador sobrecargado ++ [incrementar un vector con el vector (1,1)].

Vamos a sobrecargar el operador ++ para la clase `vector`, de modo que el vector *(3,4)* se convertirá tras la incrementación en el vector *(4,5) : [(3,4) + (1,1) = (4,5)]*.

```
#include <iostream>
using namespace std;

class vector
{
   double x,y;
   public:
   void iniciar (double a, double b)
      {x = a; y = b;}

   void visualizar()
      {
      cout << "vector x =" <<x;
      cout << "y = " << y << \n";
      {

   vector & operator++()
      {
         x++; y++;
         return *this;
      }
};
main()
{
```

```
   vector v;
   v.inicializar(10.5, 20);
   v.visualizar();        // visualizar 10.5 20

   ++v;                   // incremento del vector v en 1,1
   v.visualizar();        // visualizar 11.5 21
}
```

El operador ++ está sobrecargado y podría ser utilizado en otras partes, como es costumbre en los objetos aritméticos.

21.3.1. Versiones prefija y postfija de los operadores ++ y ––

En las declaraciones y definiciones de `operator++()` y `operator–-()`, el compilador distingue entre las versiones prefija y postfija de esos operadores, llamando a las funciones operador sin argumento o con un argumento instrumental `int`, respectivamente. La versión prefija del operador de incremento se sobrecarga mediante una función que no es miembro, definiendo una versión de ++ con un parámetro; la versión postfija se sobrecarga, sin embargo, definiendo una versión de ++ con dos parámetros, siendo el segundo argumento de tipo `int`. Obsérvese que cuando los operadores ++ y –– se sobrecargan como miembros de una clase, las versiones prefijas no tienen ningún parámetro explícito, mientras que las versiones postfijas tienen un parámetro explícito (de tipo `int`).

> La única diferencia entre las dos definiciones es el parámetro de tipo `int`,

Los dos operadores se pueden llamar con la notación usual o con la notación funcional, en cuyo caso se han de proporcionar los parámetros para distinguir entre los operadores. Así, suponiendo la clase `complejo`:

```
class complejo {
   float r, i;
public:
   complejo(float a = 0, float b = 0);
   ...
   complejo& operator++();
   complejo operator++(int);
};

complejo c;
```

El operador ++ se puede invocar con la notación estándar del operador unitario como

```
++c;
```

o bien como

```
c++;
```

o bien, en la notación de función miembro como

```
c.operator++();    // notación funcional de la versión prefija
```

o bien, como

```
c.operator++(0);   // notación funcional del operador postfija
```

El primer formato de la función miembro es la función prefija de ++ y el segundo formato representa la versión postfija. Las versiones prefija y postfija del operador decremento –– se sobrecargan de modo similar.

Consideremos una clase Fecha, que, entre otras funciones miembro y datos, contiene una función denominada `IncrementarFecha`.

```
class Fecha
{
   // ... datos privados
public:
   // constructores y funciones miembro
   // operador de incremento prefijo
Fecha& operator++()
   {
   // Llamada a la función miembro IncrementarFecha
   IncrementarFecha();
   return *this;
   }

// operador de incremento postfijo
Fecha temp = *this; operator++(int)
   {
   IncrementarFecha();
   return temp;
   }
};
```

Se puede implementar simultáneamente un operador unitario prefijo y postfijo.

```
x &operator++();
x &operator++(int);
```

21.3.2. Sobrecargar un operador unitario como función miembro

El siguiente ejemplo muestra cómo se puede sobrecargar un operador unitario prefijo como una función miembro:

```
#include <iostream>
class Unitaria {
   int x;
public:
   Unitaria()                    {x = 0;}
   Unitaria(int a)               {x = a;}
   Unitaria& operator––()        {--x; return *this;}

   void visualizar(void)         {cout << x << "\n";}
}

main()
{
```

```
   Unitaria ejemplo = 65;
   for(int i = 5; i > 0; i--)
   {
      --ejemplo;          // ejemplo.operator––();
   }
}
```

La función operador unitario ––() está declarada, y ya que es una función miembro, el sistema pasa el puntero `this` implícitamente. Por consiguiente, el objeto que se liga a la función miembro se convierte en el operando de este operador.

21.3.3. Sobrecarga de un operador unitario como una función amiga

La siguiente función se define como una función amiga:

```
#include <iostream>
class Unitaria {
   int x;
public:
   Unitaria()                {x = 0;}
   Unitaria(int a)           {x = a;}
   friend Unitaria& operator++(Unitaria y)
}
void visualizar           {cout << x << "\n";}
};

int main()
{
   Unitaria ejemplo = 65;
   for(int i = 5; i > 0, i--)
   {
      ++ejemplo;         // operator++(ejemplo);
   }
}
```

La función operador ++() está definida, y debido a que es una función amiga, el sistema no pasa el puntero `this` implícitamente. Por consiguiente, se debe pasar explícitamente el objeto de la clase `Unitaria`.

Los operadores de incremento y decremento ++ y –– tienen características especiales ya que presentan dos modalidades, una modalidad prefija (++ o –– se sitúan delante del nombre de la variable) y una modalidad postfija (donde ++ o –– se sitúan después del nombre de la variable). Ambas modalidades incrementan o decrementan su operando pero la diferencia entre ellas es que la modalidad prefija proporciona el nuevo valor del operando como resultado, mientras que la modalidad postfija proporciona como resultado el valor más antiguo del operando.

Ejemplo 21.3

Sobrecarga de operadores unitario como función miembro y como función amiga.

La función operador unitario --() está declarada, y ya que es una función miembro, el sistema pasa el puntero `this` implícitamente. Por consiguiente, el objeto que llama a la función miembro se convierte en el operando de esta operador. La función operador ++() está definida, como función amiga, por lo

que el sistema no pasa el puntero `this` implícitamente. Por consiguiente, se debe pasar explícitamente el objeto de la clase `Unitaria`.

```
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

class Unitaria {
      int x;
public:
      Unitaria()              {x = 0;}// constructor por defecto
      Unitaria(int a)         {x = a;}// constructor con un parametro
      friend Unitaria& operator++(Unitaria y) {++y.x; return y; }
      Unitaria& operator-- () {--x ; return *this;}
      void visualizar(void)   {cout << x << "\n";}
};
int main(int argc, char *argv[])
{
 Unitaria ejemplo = 65;
 for(int i = 5; i > 0; i--)
  {
      ++ejemplo;   // operator++(ejemplo);
  }
 for(int i = 5; i > 0; i--)
 {
      --ejemplo;   // ejemplo.operator--();
 }
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

Ejercicio 21.2

Crear una clase Contador que permita seguir la cuenta de sucesos que se producen en un sistema. La cuenta se lleva en una variable Cuenta, miembro dato de la clase Contador, y se trata de que mediante el operador ++ se realiza el incremento del contador.

```
// contador1.cpp
// incremento de la variable del contador con el operador ++
#include <iostream>
using namespace std;

class Contador
{
   private:
      unsigned int cuenta;
   public:                              // cuenta
     Contador()           {cuenta = 0;} // constructor
     int leer_cuenta() {return cuenta;} // devuelve cuenta
     cuenta& operator ++() {++ cuenta;return *this }
                                            // incrementa cuenta
};
```

```
   void main()
   {
      Contador C1, C2;       // definir e inicializar

      cout << "\nC1 = " << C1.leer_cuenta();
      cout << "\nC2 = " << C2.leer_cuenta();

      ++C1;    // incrementar C1 (incrementa cuenta
      ++C2;    // incrementar C2 (incrementa cuenta)
      ++C2;    // incrementar C2 (incrementa cuenta)

      cout << "\nC1 = " << C1.leer_cuenta();
      cout << "\nC2 = " << C2.leer_cuenta();
}
```

En este programa se han creado dos objetos de la clase `Contador`: `C1` y `C2`. Las cuentas de los objetos se visualizan y, a continuación, se incrementan mediante el operador ++, los objetos `C1` y `C2`, que implícitamente invocan a la función `operator++` cuya acción consiste en incrementar el valor del dato `cuenta` en 1. La salida del programa-es:

```
C1 = 0
C2 = 0
C1 = 1
C2 = 2
```

Los incrementos han sido una vez `(C1)` y dos veces `(C2)`, debido a que se ha utilizado la notación prefija.

21.3.4. Operadores de incremento y decremento

Los operadores de incremento y decremento suelen ser muy útiles en aplicaciones de tiempo o conteo. Así, por ejemplo, desarrollemos una clase `Punto` con operadores ++ y -- sobrecargados y con notaciones prefija y postfija, que realiza incrementos y decrementos de las coordenadas de cada punto.

si `p` *es* (1, 2, 3) *entonces* `++p` *es* (2, 3, 4)

```
// sobrecarga de los operadores incremento y decremento
// postfijo y prefijo
#include <iostream>
using namespace std;

class Punto {
public :
   // ...
   Punto& operator ++();            // prefija
   Punto  operator ++(int);         // postfija
   Punto& operator --();
   Punto  operator --(int);
//clase Punto
};
// ...
```

```
// operador de incremento con prefijo (++p)
inline Punto& Punto::operator ++()
{
   x += 1.0;
   y += 1.0;
   z += 1.0;
   return *this;
}
// operador de incremento con postfijo (p++)
inline Punto Punto::operator ++ (int)
{
   x += 1.0;
   y += 1.0;
   z += 1.0;
   return Punto (x - 1.0, y - 1.0, z - 1.0);
}

// operador de decremento con prefijo (--p)
inline Punto& Punto::operator ––()
{
   x -= 1.0;
   y -= 1.0;
   z -= 1.0;
   return *this;
}

// operador de decremento con postfijo (p--)
inline Punto Punto::operator --(int)
{
   x -= 1.0;
   y -= 1.0;
   z -= 1.0;
   return Punto (x + 1.0, y + 1.0, z + 1.0);
}

// programa principal
int main()
{
   Punto p(5.0, 7.0. 8.0);
   Punto q(1.0, 2.0, 3.0);
   Punto r--;
   // presentación de p
   cout << " p ( " << p.X() << " , " << p.Y() << " , "
         << p.Z() << " ) " << endl;

   r = ++p;
   cout << "++p (" << r.X() << " , " << r.Y()
         << ","      << r.Z() << " ) " << endl;

   // presentación de q
   cout << " q(" << q.X()    << " , " << q.Y()
         << " , "  << q.Z()    << " ) " << endl;
```

```
   r = q++;
   cout << "q++(" << r.X() << " , " << r.Y()
         << " , " << r.Z() << " ) " << endl;
}
```

Al ejecutar el programa se obtiene la siguiente salida en pantalla:

```
p     (5, 7, 8)
++p   (6, 8, 9)
q     (1, 2, 3)
q++   (1, 2, 3)
```

21.4. OPERADORES BINARIOS

Los operadores binarios toman dos argumentos, uno a cada lado del operador. La Tabla 21.2 lista los operadores binarios que se pueden sobrecargar.

Tabla 21.2. Operadores binarios que se pueden sobrecargar

<table>
<tr><th>Operador</th><th>Significado</th></tr>
<tr><td>+</td><td>Adición (suma)</td></tr>
<tr><td>-</td><td>Resta</td></tr>
<tr><td>*</td><td>Multiplicación</td></tr>
<tr><td>/</td><td>División</td></tr>
<tr><td>%</td><td>Módulo</td></tr>
<tr><td>=</td><td>Asignación</td></tr>
<tr><td>+=</td><td>Suma con asignación</td></tr>
<tr><td>-=</td><td>Resta con asignación</td></tr>
<tr><td>*=</td><td>Multiplicación con asignación</td></tr>
<tr><td>/=</td><td>División con asignación</td></tr>
<tr><td>%=</td><td>Módulo con asignación</td></tr>
<tr><td>&</td><td>AND bit a bit</td></tr>
<tr><td>|</td><td>OR bit a bit</td></tr>
<tr><td>^</td><td>OR exclusivo bit a bit</td></tr>
<tr><td>^=</td><td>OR exclusivo bit a bit con asignación</td></tr>
<tr><td>%=</td><td>AND bit a bit con asignación</td></tr>
<tr><td>|=</td><td>OR bit a bit con asignación</td></tr>
<tr><td>==</td><td>Igual</td></tr>
<tr><td>!=</td><td>No igual</td></tr>
<tr><td>></td><td>Mayor que</td></tr>
<tr><td><</td><td>Menor que</td></tr>
</table>

Tabla 21.2. Operadores binarios que se pueden sobrecargar *(continuación)*

Operador	Significado
`>=`	Mayor o igual que
`<=`	Menor o igual que
`||`	OR lógico
`&&`	AND lógico
`<<`	Desplazamiento a izquierda
`<<=`	Desplazamiento a izquierda con asignación
`>>`	Desplazamiento a derecha
`>>=`	Desplazamiento a derecha con asignación
`->`	Puntero
`->*`	Puntero a miembro

21.5. SOBRECARGA DE OPERADORES BINARIOS

Los operadores binarios pueden ser sobrecargados tan fácilmente como los operadores unitarios. Se pueden sobrecargar pasando a la función dos argumentos si no es función miembro. El primer argumento es el operando izquierdo del operador sobrecargado y el segundo argumento es el operando derecho, si la función no es miembro de la clase. Consideremos un tipo de clase `t` y dos objetos `x1` y `x2` de tipo `t`.

Definamos un operador binario + sobrecargado. Entonces:

```
x1 + x2
```

se interpreta como

```
operator+(x1, x2)
```

o como

```
x1.operator+(x2)
```

Un operador binario puede, por consiguiente, ser definido:

- bien como una función de dos argumentos;
- bien como una función miembro de un argumento, y es el caso más frecuente;
- pero nunca las dos a la vez.

21.5.1. Sobrecarga de un operador binario como función miembro

El siguiente ejemplo muestra cómo sobrecargar un operador binario como una función miembro:

```
#include <iostream.h>

class Binario {
   int x;
```

```
public:
   Binario()        {x =0;}
   Binario(int a)   { x = a;}
   Binario operator + (const Binario&)const;
   void visualizar  {cout << x << "\n";}
};

Binario Binario::operator+(constBinario& a)const
{
   Binario aux;
   aux.x = x + a.x;
   return aux;
}

main()
{
   Binario primero(2.8), segundo(4.5), tercero;
   tercero = primero + segundo;
   tercero.visualizar();
}
```

La salida del programa es

```
7.3
```

Ejemplo de sobrecarga de un operador binario

Definamos un tipo de clase `vector` constituido por:

- Dos datos-miembro privados (coordenadas de tipo `double`).
- Dos funciones `public`, de los cuales:
  - — Una función miembro de inicialización.
  - — Una función miembro de visualización de coordenadas.
- Un operador sobrecargado `*` producto escalar de dos vectores.

```
#include <iostream>
using namespace std;

class vector
{
   double x, y;
   public:
      void iniciar (double a, double b)
         {x = a; y = b;}
      void visualizar()
         {
         cout << " componente x= " << x << "componente y="
         << y << endl;
         }
   double operator * (const vector v)const
      {
         return (x * v.x + y * v.y);
```

```
      }
};

main()
{
   vector v1, v2
   v1.iniciar(1,2); v2.iniciar(2,3);
   v1.visualizar(); v2.visualizar();
   cout << " su producto escalar = ";
   cout << v1*v2;   // visualiza 8
}
```

El operador multiplicación * está sobrecargado en nuestra clase `vector`. La escritura `v1 * v2` tiene ahora el sentido de la escritura natural. El resultado es un número real que corresponde al producto escalar definido en matemáticas.

De hecho, `operator*` se define como una función miembro que tiene un argumento de tipo vector y que devuelve una expresión de tipo double. La escritura `v1 * v2` es preferible a la escritura `v1.operator*(v2)`. El operador * se ha sobrecargado para un producto escalar, pero el operador multiplicación natural siempre se puede utilizar escribiendo

```
i = j* k
```

donde los objetos `i, j, k` son aritméticos (`int, long, double,...`).

El operador idóneo se elegirá automáticamente, en función del tipo de operador.

Otro ejemplo de sobrecarga de un operador binario

En C++, el operador + no se puede utilizar normalmente para concatenar cadenas, como sucede en algunos lenguajes, tales como BASIC. Es decir, en BASIC se puede hacer

```
cad3 = cad1 + cad2;
```

donde `cad1`, `cad2` y `cad3` son variables de cadena (array de tipo char), tal como «`Hola`» y «`Mundo`», y la expresión `suma` anterior *(concatenación)* producirían «`Holamundo`» en `cad3`. Sin embargo, esta operación no se puede hacer directamente en C++.

Sin embargo, si utilizamos una clase propia llamada cadena, como se muestra en el siguiente programa, se puede sobrecargar el operador + para obtenerla concatenación.

```
// archivo CADENA.CPP
#include <iostream>
#include <string>  // necesario para strcat(),strcpy() y strlen()

class cadena
{
      char * ch;
      int longitud;
   public:
      cadena () {ch = 0;}
      cadena (const char*);
      ~cadena();
      cadena operator + (const cadena&)const;
      void visualizar() {cout << ch;}
```

```
};

cadena::cadena (const char*s)
   {
   longitud = strlen(s);
   ch = new char[longitud + 1];
   strcpy (ch, s);
   }

cadena::cadena~() { delete ch;}

cadena cadena::operator + (const cadena x)const
   {
   cadena temp;
   temp.longitud = longitud + x.longitud;
   temp.ch = new char [temp.longitud + 1];
   temp.ch = strcat (ch, x.ch);
   return temp;
}

main()
{

      cadena a("Hola");
      cadena b("Mundo");
      cadena c = a+b;  // se concatena en c-: "Hola Mundo"
      a.visualizar();
}
```

Se ha definido una clase `cadena` que contiene:

- Dos miembros dato:
  - Un puntero a la cadena.
  - Una longitud igual a la longitud de la cadena más uno.
- Un constructor que reserva memoria en la memoria dinámica (memoria libre o montículo).
- Un destructor que, en el momento de la destrucción del objeto, libera la memoria libre.
- Un operador de concatenación.
- Una función de visualización.

Los objetos `a` y `b` están definidos, y la escritura

```
a + b
```

significa simplemente que la cadena `b` ha sido *pegada* a la cadena `a`.

Sólo se realiza la concatenación si hay espacio en `a` para pasar las dos cadenas. Otra alternativa sería copia `b` en `a` hasta completar 100 caracteres, y en cualquier caso, hacer que el operador + devuelva *true* si se ha podido realizar con éxito la operación, o *false* en caso contrario. Haciéndolo de este modo, la concatenación se haría con:

```
if (a + b) a.visualizar();
else count << "fallo\n";
```

21.5.2. Sobrecarga de un operador binario como una función amiga

La siguiente función operador + se define como una función amiga:

```
#include <iostream>

class Binario {
   int x;
public:
   Binario()        { x = 0; }
   Binario(int a)   { x = a; }
   friend Binario operator+(const Binario&,const Binario&);
   void visualizar(void) {cout << x << "\n";)
};

Binario operator+(const Binario& a, const Binario& b)
{
   Binario aux;
   aux.x = a.x + b.x;
   return aux;
}
main()
{
   Binario primero(2.8), segundo(4.5), tercero;
   tercero = primero + segundo;
   tercero.visualizar() ;
}
```

La salida del programa es

```
7.3
```

La función operador binario +() está declarada; debido a que es una función amiga, el sistema no pasa el puntero `this` implícitamente, y por consiguiente se debe pasar el objeto `Binario` explícitamente con ambos argumentos. Como consecuencia, el primer argumento de la función miembro se convierte en el operando izquierdo de este operador y el segundo argumento se pasa como operando derecho.

21.6. OPERADORES + Y –

En C++, si `Punto` es una clase, es posible escribir expresiones de la forma

```
Punto P1, P2, P3;
// ...
P3 = P1 + P2;
```

siempre que el operador + esté sobrecargado.

Supongamos un punto arbitrario `P (Px,Py,Pz)` en un sistema de coordenadas de tres dimensiones *(x, y, z)*.

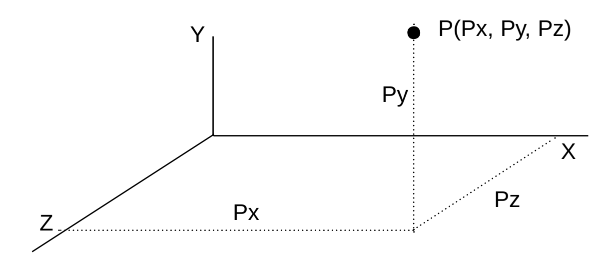


Figura 21.1. Un punto P en un sistema de coordenadas cartesianas de tres dimensiones (x, y, z).

Las operaciones suma y resta se realizan de la forma siguiente: para los puntos P1, P2 y P3

```
P1 = (4, 5, 7)
P2 = (2, 1, 4)
P3 = (1, 5, 2)

P5 = P1 + P2 = (6, 6, 11)
P6 = P1 - P2 = (2, 4, 3)
```

Ejercicio 21.2

Sobrecargar los operadores + y – para la clase Punto.

```
// sobrecar1.cpp
// sobrecarga de los operadores aritméticos + y - para la clase
// Punto
#include <iostream>
using namespace std;

// clase Punto
class Punto {
private:
   // miembros privados dato
   double x, y, z;
public:
   // constructores
   Punto(): x(0.0), y(0.0), z(0.0) {}
   Punto(double arg_x, double arg_y, double arg_z = 0.0)
      : x (arg_x),y (arg_y), z (arg_z) {}
   // funciones miembro públicas
   double& X() {return x;}
   double& Y() {return y;}
   double& Z() {return z;}
   const double& X() const {return x;}
   const double& Y() const {return y;}
   const double& Z() const {return z;}

   // operadores sobrecargados
   Punto operator+ (const Punto& p)const;
   Punto operator- (const Punto& P)const;
};    // clase Punto
```

```
// operadores sobrecargados

// p1 + p2

inline Punto Punto::operator+ (const Punto& p)
{
   return Punto (x + p.x, y + p.y, z + p.z);
}

// p1-p2

inline Punto Punto::operator- (const Punto& p)
{
   return Punto (x - p.x, y - p.y, z - p.z);
}

void main()
{
   // definir objetos
   Punto p1 (4.0, 5.0, 7.0);
   Punto p2 (2.0, 1.0, 4.0);
   Punto p3, p4, p5;

   // sumar p1 y p2
   p3 = p1 + p2;
   p4 = p1 - p2;
   p5 = p1 + p2 - p3 - p4;

   cout << " p3( "  << p3.X() << " , " << p3.Y()
      << " , "      << p3.Z() << " ) " << endl;
   cout << " p4 ( " << p4.X() << " , " << p4.Y()
         << " , "   << p4.Z() << " ) " << endl;
}
```

Al ejecutar el programa se tendrá la salida

```
p3 (6, 6, 11)
p4 (2, 4, 3)
```

Sintaxis

La sintaxis general de una función miembro operador sobrecargada es:

especificador_tipo NombreClase: **operator** op (*lista_parámetros*)

op es el operador sobrecargado (`<, +, =,[], ...`).

21.7. SOBRECARGA DE OPERADORES DE ASIGNACIÓN

Por defecto, C++ sobrecarga el operador de asignación, =, para objetos de clases definidas por el usuario. Por ejemplo, la siguiente asignación

```
p4 = p1;
```

copiará *exactamente* (miembro a miembro) todos los miembros datos de `p1` a `p4`, sin tener que sobrecargar explícitamente el operador de asignación para la clase `Punto`. Cuando se asigna un objeto a otro objeto, se hace una copia miembro a miembro de forma opuesta a como se hace la copia bit a bit (*bitwise*).

La asignación de un objeto de una clase a otro de esa clase se realiza utilizando el operador de asignación de copia. Los mecanismos son esencialmente los mismos que los citados de inicialización por defecto miembro a miembro, pero hace uso de un operador de asignación de copia implícito en lugar del constructor de copia. El formato general del operador de asignación de copia es el siguiente:

```
// formato general del operador de asignación de copia
const nombreClase& nombreClase::operator=(const nombreClase &v)
{
      // evita autoasignaciones
      if (this!= &v)
      {
            // semántica de la copia de clases
      }
      // devolver el objeto asignado
      return *this;
}
```

donde el test condicional

```
if (this!= &v)
```

evita la asignación de un objeto de una clase a sí misma.

Reglas[1]

Cuando un objeto de una clase se asigna a otro objeto de su clase, tal como

```
nuevoObj = antiguoObj,
```

se siguen los siguientes pasos:

1. Se examina la clase para determinar si se proporciona un operador de asignación de copia explícita.
2. Si es así, se comprueba su nivel de acceso para determinar si se puede invocar o no, dentro de una porción del programa.
3. Si no es accesible, se genera un mensaje de error en tiempo de compilación; en caso contrario, se invoca para ejecutar la asignación.
4. Si no se proporciona una instancia explícita, se ejecuta la asignación miembro a miembro.
5. Bajo la asignación miembro a miembro por defecto, cada miembro dato de un tipo compuesto o incorporado se asigna al valor de su miembro correspondiente.
6. A cada miembro del objeto de la clase se aplica recursivamente los pasos 1 a 6 hasta que se asignan todos los miembros dato de los tipos compuestos o incorporados.

[1] Lipman, S. y Lajoie, J., C++ Primer, 3.ª ed., Reading, Masachussets, Addison-Wesley, 1998, p. 730.

Ejemplo 21.4

Sobrecarga del operador de asignación de la clase Punto.

```
#include <iostream>
using namespace std;

// clase Punto
class Punto
{
private :
   double x, y, z;
public:
   Punto(): x(0.0), y(0.0), z(0.0) {}
   Punto (double x_arg, double y_arg, double z_arg = 0.0)
      : x(x_arg), y(y_arg), z(z_arg) {}
   // funciones miembro públicas
   // ...
   // operador sobrecargado
   const Punto& operator = (const Punto& p);
};    // clase Punto
// operador de asignacion sobrecargado
const Punto& Punto::operator = (const Punto& p)
{
   x = p.x;
   y = p.y;
   z = p.z;
   cout << " el operador = sobrecargado se ha llamado" << endl;
   return *this;
}
// Una funcion main que utiliza el operador =
int main()
{
   // definir objetos
   Punto p1 (2, 4,6), p2;

   cout << " p1( " << p1.X() << " , " << p1.Y()
        << " , "  << p1.Z() << " ) ″ << endl ;
   cout << "p2( "  << p2.X() << " , " << p2.Y()
        << " , ″  << p2.Z() << " ) " << endl;
   p2 = p1-;     // asignación
   cout << " p2( " << p2.X() << " , " << p2.Y()
        << " , "  << p2.Z() << " ) " << endl;
}
```

Al ejecutar este programa se visualizará

```
p1(2, 4, 6)
p2(0, 0, 0)
el operador = sobrecargado se ha llamado
p2(2, 4, 6)
```

La definición de la función miembro operador de asignación sobrecargado indica un tipo de retorno `Punto` y por consiguiente es posible el encadenamiento de operadores de asignación:

```
// ...
p1 = p2 = p3 = p4;      // asignación encadenada
```

21.7.1. Sobrecargando el operador de asignación

Hasta ahora se ha utilizado el operador de asignación = con su operación incorporada. Siempre que se hace una asignación en una expresión se realiza una operación de copia. Por ejemplo:

```
int x = 123;
int y = 24;
int z;

z = x + y;
```

deja la variable `z` con una copia del número que resulta de evaluar la expresión del lado derecho del operador. Esta operación se realiza también incluso para clases. Así, por ejemplo, suponiendo una clase `string`, el segmento de código

```
string s("Esto es sólo una prueba");
string t("Todo es correcto");
string u;

u = s + t;
```

produce un objeto `u` que contiene una copia del objeto resultante de la operación del operador, suma aplicada a los objetos clase `string`.

El operador de asignación = sólo puede ser sobrecargado declarando una función miembro no estática. Por ejemplo:

```
class string {
   ...
   string& operator = (const string& str);
   ...
   string (const string&);
   string();
};
```

Este código, con definiciones adecuadas de `string::operator=()`, permite asignaciones de cadenas (string) `str1 = str2`, tal como en otros lenguajes. Al contrario de otras funciones operador, la función operador asignación no puede ser heredada por clases derivadas. *Si, para cualquier clase x, no existe operador = definido por el usuario, el operador = se define por defecto como una asignación miembro a miembro de los miembros de la clase x:*

```
const x& x::operator = (const x& fuente)
{
   // asignación de miembros
}
```

El siguiente listado muestra la clase `string` con un operador de asignación sobrecargado. En este caso, el operador redefinido permite la asignación de una cadena de caracteres tradicional C++ (`char*`) directamente a un objeto de una clase `string`. Es lo que se conoce como operador de asignación-conversión

```
#include <iostream>
using namespace std;

#include <string.h>

class string {
   char * v;
   int lon;
public:
   string() {lon = 0-; new char[10];}
   string(char *);
   ~string() {delete v;}

   string operator + (string);
   string operator + (char*);
   const string& operator += (const string&);

   const string operator = (const char*);
   char * mostrar() {return v;}
   };

   string::string(char*str) {
      lon = strlen(str) + 1;
      v = new (char[lon]);
      strcpy(v,str);
   }

   const string& string::operator = (const char*cad)
   {
      lon = strlen(cad) + 1;
      delete v ;
      v = new char [lon];
      strcpy (v,cad);
      return *this;
   }

   int main()
   {
      string u("Esto es sólo un test");

      u = "Esto no es un test";
      cout << "u = " << u.mostrar() << "\n";

   }
```

21.7.2. Operadores como funciones miembro

El objetivo de tener acceso a los miembros privados de una clase (por ejemplo, `vector`) se puede también obtener incluyendo la función `operator+` como miembro de esa clase. El número de parámetros

recibidos por una función operador miembro sobrecargada depende del tipo de operador. Los operadores unitarios no tienen argumentos, los operadores binarios tienen un argumento (el operador ternario-? no se puede sobrecargar).

```
// Operador miembro: Función operador como miembro de la clase
#include <iostream>
using namespace std;

class vector {
public-
   vector(float xx = 0, float yy = 0):x(xx), y(yy) {}
   void imprimirvec() const;
   vector operator + (const vector &b)const;
private:
   float x, y;
};

void vector imprimirvec() const
{
   cout << x << ´,´ << y << endl;
}

vector vector::operator+(const vector &b)const
{
   return vector (x + b.x, y + b.y);
}

int main()
{
   vector u(3, 1), v(1, 2), s;
   S = u + v;                // calcular la suma de dos vectores
   S.imprimirvec();          // salida (4,3)
   return 0;
}
```

En la definición de la función `operador+` se aprecia que `x` e `y` se refieren al primer operando; es decir, a la variable vector `u`. La definición del operador sólo muestra el parámetro `b`, que corresponde al segundo operando `b`. El primer operando es *implícito* ya que la sentencia

```
S = u + v;
```

se considera como una forma abreviada de

```
S = u.operator+(v);
```

21.7.3. Operador [ ]

El operador de subíndice binario `[]` se puede definir para clases que representan una abstracción de contenedor y de la cual se recuperan elementos individuales. Este operador se utiliza normalmente para índices de arrays. Realmente el operador subíndice realiza una función útil: oculta la aritmética de punteros. Así, por ejemplo, si se tiene el array:

```
char nombres[20];
```

y se ejecuta una sentencia tal como:

```
ch = nombres[12];
```

el operador [] hace que la sentencia de asignación añada 12 a la dirección base del array nombres para localizar el dato almacenado en esa dirección de memoria (el proceso se ilustra en la Figura 21.2).

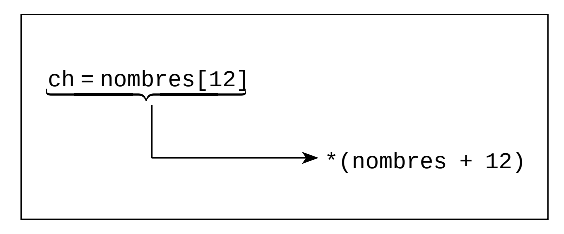


Figura 15.2. Operación de subíndices.

El operador [] se puede sobrecargar sólo mediante funciones miembro y requiere dos operandos ya que es un operador binario. Así, sea la expresión

```
p = x[i];
```

Aquí, [] es el operador, x es el primer argumento y i es el segundo argumento. Dicho de otro modo, el primer argumento es el objeto que se indexa y el segundo argumento es el índice.

En C++ se puede sobrecargar también este operador y proporciona muchas extensiones útiles del concepto de subíndices de arrays. [] se considera un operador binario porque tiene dos argumentos. En el ejemplo

```
p = x[i];
```

los argumentos son x e i. La función operador correspondiente es operator[]; ésta puede ser definida por el usuario para una clase x, se interpreta como x.operator[](y).

```
X op Y ↔ X.operator op(Y)
```

o dicho de otro modo

```
X[i] equivale a X.operator[](i)
```

21.7.4. Sobrecargando el operador de llamada a funciones ()

Uno de los operadores más usuales, el operador de llamada a función(), puede ser sobrecargado sólo mediante funciones miembro. La llamada a función se considera como un operador binario.

```
Expresión principal (<lista de expresiones)
```

donde *expresión principal* es un operando y *lista de expresiones* (que puede ser vacía) el otro operando.

La función operador correspondiente es `operator()` y puede ser definida por el usuario para una clase `t` (y cualquier clase derivada) sólo mediante una función miembro no estática. Las siguientes expresiones son equivalentes:

```
x(i)             equivale a    x.operator() (i)
x(arg1, arg2)    equivale a    x.operator() (arg1, arg2)
```

21.8. SOBRECARGA DE OPERADORES DE INSERCIÓN Y EXTRACCIÓN

Las sentencias de flujos se utilizan para entradas y salidas, lectura y visualización de valores. Observe que los flujos no son parte del lenguaje C++, pero se implementan como clases en la biblioteca de C++. Las declaraciones para estas clases se almacenen en el archivo de cabecera `<iostream>`. En C++ es posible sobrecargar los operadores de inserción y extracción de modo que pueda manipular cualquier sentencia de flujo que incluya cualquier tipo de clase. Como se definen en el archivo de cabecera `<iostream>`, estos operadores trabajan con todos los tipos predefinidos, tales como `int, long, double` y `char*`. Cuando se definen sus propias clases, tales como la clase `string`, se puede desear sobrecargar las definiciones de los operadores << y >> de modo que trabajen con sus clases. Por ejemplo, una vez que se sobrecarga el operador >>, se pueden leer caracteres de un flujo de entrada almacenándolos en un objeto de una clase `string` escribiendo:

```
string entrada_usuario;
cin >> entrada_usuario;
```

De modo similar, para visualizar una cadena `string` se escribirá lo siguiente:

```
string saludos = "Hola, Mundo";
cout << saludos << endl;
```

> Sobrecargando los operadores de flujos de entrada y salida, éstos pueden manipular, además de los tipos de datos predefinidos (`int`, `long`, `double`, `char*`, etc.), cualquier tipo de objetos de clases.

21.8.1. Sobrecarga de flujo de salida

La sobrecarga del operador de flujo de salida << puede añadir a sus propias clases los tipos de datos que diseñan las sentencias de flujo de salida.

El listado `punto.cpp` muestra un ejemplo típico. El programa declara una clase llamada `punto` que almacena dos valores enteros `x` e `y`, que representan los valores de las coordenadas de una posición en una pantalla de gráficos. Normalmente, para visualizar los valores de miembros privados como `x` e `y` habrá que llamar a funciones miembro tales como `leerx` y `leery`. Sin embargo, añadiendo a la clase `punto` un operador de flujo de salida << sobrecargado, se podrán escribir objetos `punto` en sentencias de flujo de salida.

```
//punto.cpp

#include <iostream>

class punto{
private:
```

```
   int x, y;
public:
   punto()                      {x = y = 0;}
   punto (int xx, int yy)       {x = xx; y = yy;}
   void fijarx(int xx)          {x = xx;}
   void fijary(int yy)          {y = yy;}
   int leerx(void)              {return x;}
   friend ostream& operator << (ostream& os, const punto &p);
};

int main()
{
   punto p;

   cout << p << '\n';
   p.fijarx(50);
   p.fijary(100);
   cout << p << '\n';
   return 0;
}
ostream& operator << (ostream& os, punto &p)
{
   os << "x = " << p.x << ", y = " << p.y;
   return os;
}
```

La última línea de la clase `Punto` sobrecarga el operador de flujo de salida declarando una función miembro amiga `operator <<()`. Esta función miembro amiga devuelve `ostream&` y declara dos parámetros `os` de tipo `ostream` y `p`, una referencia a un objeto de Punto.

Normalmente, las funciones operador sobrecargadas siempre son amigas (`friend`) de la clase (aunque no es un requisito estricto). Si `operator<<()` no fuera amiga, no podría acceder a los miembros dato privados, a menos que fueran funciones miembro.

La ejecución del programa visualiza

```
x = 0,y = 0
x = 50, y =100
```

Nota

Es posible poner en cascada múltiples objetos en una sentencia de flujo de salida.

```
Punto p1, p2, p3 ;
// ...
cout << p1 << ":" << p2 << ":" << p3 ;
```

21.8.2. Sobrecarga de flujo de entrada

La sobrecarga de flujo de entrada >> es similar a la sobrecarga del flujo de salida. El programa `puntouno.cpp` mejora el listado `punto.cpp` añadiéndole una función operador de sobrecarga de entrada.

```
//puntouno.cpp
#include <iostream>
using namespace std;

class Punto {
private:
   int x, y;
public:
   punto()                    {x = y = 0;}
   punto (int xx, int yy)     {x = xx; y = yy;}
   void fijar(int xx)         {x = xx;}
   void fijary(int yy)        {y = yy;}
   int leerx(void)            {return x;}
   int leery(void)            {return y;}
   friend ostream& operator << (ostream& os, const punto &p);
   friend istream& operator >> (istream& is, punto &p);
};

int main() {
   punto p;
   cout << p << '\n';
   p.fijarx(50);
   p.fijary(100);
   cout << p << '\n';
   cout << " Introducir valores x e y = ";
   cin >> p;
   cout << '\n Se ha introducido: " << p;
   return 0;
}

ostream& operator<< (ostream& os, const punto &p}
{
   os << " x = " << p.x << ", y = " << p.y;
   return os;
}

istream& operator>>(istream& is, punto &p}
{
   is << p.x << p.y;
   return is-;
}
```

Otro ejemplo de sobrecarga del operador de entrada puede ser el siguiente. Considere la clase `Fecha`:

```
class Fecha{
   int dia;
   int mes;
   int anyo;
public:
   friend istream & operator >> (istream & en, Fecha &f);
};
```

Obsérvese que el operador de entrada sobrecargado, como ya se utilizó en el ejemplo anterior, utiliza la clase de entrada de flujo llamada `istream` (definida en el archivo `iostream.h`). Una implementación de la función `operator>>()` puede ser ésta:

```
istream & operator>>(istream & en, Fecha &f)
{
   cout << endl;
   cout << "¿Número de mes ?";
   en >> f.mes                  // obtiene mes de teclado
   cout << "¿Número de dia ?";
   en >> f.dia                 ;// obtiene dia de teclado
   cout << "¿Número de anyo ?";
   en >> f.anyo;                // obtiene anyo de teclado
   return en;                   // apila el operador >>
}
```

En un programa, todo lo que se debería hacer para obtener una fecha sería ejecutar esta sentencia:

```
cin >> hoy;
```

Otro ejemplo de una función operador de entrada sobrecargado podría ser éste:

```
istream & operator >> (istream & en, Persona &p)
{
   cout << "¿Cuál es el nombre del empleado?";
   en >> p.nombre;
   cout << " ¿Cuál es ″ << p.nombre << " su edad? ";
   en >> p.edad;
   cout << " ¿Cuál es " << p. nombre << " Su salario? ";
   en >> p.salario;
   return en;
}
```

21.9. CLASE CADENA

La clase `string.h` utiliza funciones amigas (*friend*) para sobrecargar los operadores relacionales cuando un objeto cadena se compara a una cadena literal. Sin embargo, utiliza funciones miembro cuando un objeto cadena se compara con otro objeto de cadena. Los operadores restantes (asignación, conversión de tipos y acceso a caracteres individuales) se sobrecargan como funciones miembro.

21.9.1. Clase cadena (`string`)

C++ proporciona una clase que representa cadenas, denominada `string` y cuyo archivo de cabecera es *cstring.h*, pero dado su carácter didáctico y la facilidad para asimilar los conceptos de sobrecarga de operadores, vamos a definir una clase `String` que encapsula un miembro dato cadena con una longitud máxima de 80 caracteres.

```
// string.cpp
// clase cadena (string)
#include <iostream>     // Entrada/salida
using namespace std;
```

```
#include <string>       // función strcpy()
#include <cstdlib>       // función exit()

//Longitud máxima de la cadena
const int LONG_CAD = 80;
class String {
private:
   char string[LONG_CAD];
public:
   // constructor: convierte una cadena C++ a String y asigna
   // por defecto una cadena nula
   String (char S[] = "\0")
      {strcpy (string, s); }
   // función miembro
   void Mostrar() {cout << string << endl-;}
   // operadores sobrecargados (un objeto String se puede asignar a
   // otro objeto String)
   String& operator = (const String& S);
   String& operator = (char S[]);
   String  operator + (const String& S);
   String  operator + (char S[])const;
};    // clase String

// operadores sobrecargados =
// String a String
String String::operator = (const String& S)
{
   strcpy (string, S.string);
   return String (string);
}

// cadena C++ a String
String String::operator = (char S[])
{
   strcpy (string, S);
   return String (string);
}

// Sobrecarga de operador +
// String + String
String String::operator + (const String & S)
{
   if ((strlen (string) + strlen (s.string))> LONG_CAD)
   {
      cout << " cadenas concatenadas demasiado largas " << endl;
      exit (EXIT_SUCCESS);
   }
   String t;

   strcpy (t.string, string);        // cadena a t
   strcat (t.string, s.string);      // sumar dos cadenas
   return t;
}
```

```
// cadena C++ + String
   String String::operator + (char S[])
   {
      if ((strlen (string) + strlen (S)) > LONG_CAD)
      {
         cout << " cadenas concatenadas demasiado larga " << endl;
         exit (EXIT_SUCCESS);
      }
      String t;

      Strcpy (t.string, string);
      strcpy (t.string, s);
      return t;
   }

   void main()
   {
      String S1;       // cadena nula
      S1.Mostrar();

      String S2 ("Sierra");
      S2.Mostrar();

      String S3 = " Magina ";
      S3.Mostrar();

      String S4 = S2 + S3 + "es muy bonita";  // concatenar
      S4.Mostrar();
   }
```

La ejecución del programa produce

```
Sierra
Magina
Sierra Magina es muy bonita
```

Existen tres formas de operadores sobrecargados:

Funciones miembro.
Funciones amigas (`friend`) de la clase.
Funciones globales.

21.9.2. Funciones amigas

En numerosas ocasiones se necesita poder acceder a los miembros privados de una clase, pero normalmente el operador actúa como función global y no como función miembro de una clase, por lo que no puede referirse a los miembros dato privados de la clase directamente. En estos casos, es posible declarar los operadores como *amigos* (`friend`) de la clase. La declaración de una función u operador como amiga garantiza que esa función u operador accede a sus miembros no públicos.

Una declaración amiga comienza con la palabra reservada `friend`. Puede aparecer dentro de una definición de la clase. Aunque las funciones amigas se declaran dentro de una clase, no son funciones

miembro, por lo que no están afectadas por la sección pública, privada o protegida en que están declarados dentro del cuerpo de la clase.

Ejemplo 21.5

```
class String {
   friend bool operator==(const String &, const String&);
   friend bool operator==(const char*, const String&);
   friend bool operator==(const String&, const char *);
public:
   // resto de la clase String
};
// Amiga
#include <iostream.h>

class vector {
public:
   vector(float xx = 0, float yy = 0) : x(xx), y(yy) {}
   void imprimirvec() const;
   friend vector operator+ (const vector &a, const vector &b);
private:
   float x, y;
}

void vector::imprimirvec() const
{
   cout << x << ',' << y << endl;
}

vector operator+ (const vector &a, const vector &b)
{
   return    vector(a.x + b.x, a.y + b.y);
}

int main()
{
   vector u(3,1), v(1,2), S;
   S = u + v;
   S.imprimirvec();
   return 0;
}
```

Gracias al prefijo `friend` de la declaración de la función operador +, se puede acceder directamente a los miembros `x` e `y` por esta función. Aunque una función amiga no es miembro de la clase —como ya se ha comentado— se puede definir dentro de la clase.

Ejemplo 21.6

```
class vector {
public:
   vector(float xx = 0, float yy = 0): x(xx), y(yy) {}
```

```
void imprimirvec() const;
friend vector operator+ (const vector &a, const vector &b)
{
  float xa, ya, xb, yb;
  return vector (a.x + b.x, a.y + b.y);
}
```

Nota

En una clase A, se puede especificar que todas las funciones miembros de una clase B son funciones amigas, declarando la clase B como amiga de A. Se escribe así:

```
friend class B
```

APLICACIÓN

Examinar la sobrecarga de operadores utilizando un tipo de dato vector. Supongamos dos vectores:

u = (xu, yu)

V = (XV, YV)

La suma de los vectores

S = (Xs, ys)

se puede calcular así:

xs = xu + yv

Ys = yu + jv

Gráficamente la suma de los vectores *u = (3,1)* y *v = (1,2)* se ilustra en la Figura 21.3. Mostrando el vector suma *S = (4,3)*.

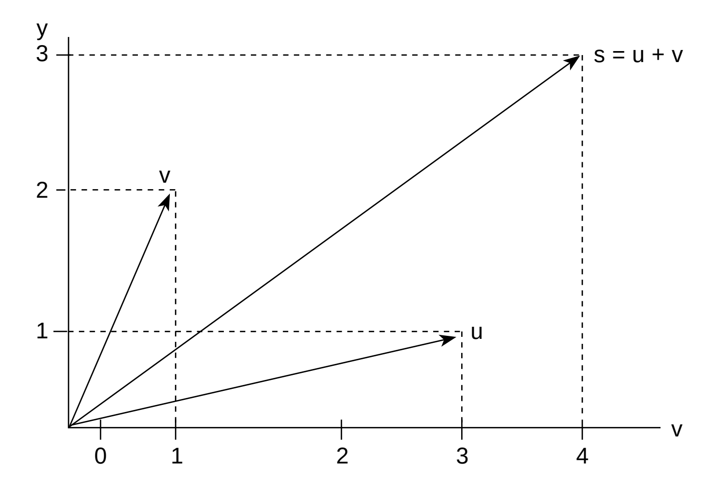


Figura 21.3. Suma de dos vectores.

La sobrecarga del operador + para vectores permitirá escribir

```
s = u + v;       // u, v y s son vectores

// programa de aplicación de la sobrecarga del operador +
#include <iostream.h>

class vector {
public:
   vector (float xx = 0, float yy = 0)-: x(xx), y(yy) {}
   vector imprimirvec() const;
   void leervec (float &xx, float &yy) const (xx = x; yy = y; }
private:
   float x, y;
};

void vector::imprimirvec() const
{
   cout << x << ',' << y << endl;
}

vector operator+ (const vector &a, const vector &b)
{
   float xa, ya, xb, yb;
   a.leervec(xa, ya)-; b.leervec(xb, yb);
   return vector(xa + xb, ya + yb);
}
int main()
{
   vector u (3,1), v(1,2), s ;
   s = u + v ;            // suma de los dos vectores
   s.imprimirvec() ;      // salida (4,3)
   return 0;
}
```

Función suma **versus** ***operador*** `suma`

Función suma	`vector suma (vector &a, vector &b)`
Llamada función suma	`s = suma (u, v) ;`
Operador suma	`vector operator + (const vector &a, const vector &b)`
Llamada operador suma	`s = u + v`

21.10. SOBRECARGA DE `NEW` Y `DELETE`: ASIGNACIÓN DINÁMICA

En las primeras versiones de C++ de AT&T no se podían sobrecargar los operadores de asignación y liberación de almacenamiento dinámico `new y delete`. En estas versiones, para controlar el almacenamiento dinámico se tenía que seguir un proceso delicado a asignar un valor al puntero `this`. Afortunadamente, esta operación ya no es necesaria, y se puede tomar el control completo de asignación y liberación de memoria utilizando para una clase sus propios operadores `new` y `delete`. Las nuevas

versiones de C++, a partir de la 2.0, ya pueden sobrecargar `new` y `delete`, y esta sobrecarga permite controlar la asignación y liberación de memoria de un modo más eficiente.

21.10.1. Sobrecarga de new

El operador predefinido `new` proporciona asignación dinámica adecuada en el montículo (*heap*). Sin embargo, en ocasiones puede ser necesario implementar una gestión dinámica de memoria a su medida que adecue las necesidades de su aplicación.

El operador `new` puede ser sobrecargado en C++. El compilador C++ considera automáticamente el operador `new` como `static`, ya que el operador se llama con frecuencia para crear instancias de clase.

Para sobrecargar `new`, inserte un prototipo de la forma:

```
void * operator new(size_t tamaño);
```

en donde el primer parámetro debe ser `size_t`, un tipo declarado en el archivo de cabecera *stdef.h*, y el operador debe devolver un puntero `void*`. El argumento para el primer parámetro es el nombre de una clase o tipo de dato. Los usos futuros de `new` para asignar espacio a objetos de clase se dirigirán a las funciones sobrecargadas. La función devolverá la dirección del espacio asignado para el objeto. Si no existe espacio disponible, la función devolverá 0.

Considere, por ejemplo, la siguiente declaración de clase:

```
class Z
{
   public:
      z() {}
      static void* operator new (size_t tamanyo);
};

main()
{
   Z* z = new Z;
   // otras sentencias
   delete z;
   return 0;
}
```

El nombre de la clase `Z` que aparece después del operador `new` es el argumento para el tamaño del parámetro. El compilador traduce el nombre de la clase en su tamaño y pasa la información al parámetro `tamanyo`.

C++ permite declarar parámetros adicionales que se puedan requerir por el operador `new`. Dado que el operador `new` se considera `static` por el compilador, no puede acceder a miembros dato no estáticos. El siguiente ejemplo contiene dos versiones del operador sobrecargado `new`.

```
class Matriz
{
   protected;
      unsigned MFila;
      unsigned Mcolumna;
      double* Ptrdatos;

public:
```

```
   Matriz(unsigned Filas, unsigned Columnas)
   {
      Mfilas = Filas;
      Mcolumnas = Columnas;
      Ptrdatos = new double [Mfilas *Mcolumnas];
   }
   static void* operator new(size_t tamanyo);
   static void* operator new(size_t, unsigned Filas, unsigned Columnas);
};

main()
{
   Matriz *mat = new Matriz(15,25);
   // otras sentencias
   delete mat;
   return 0;
}
```

En este programa se muestran las dos versiones del operador `new` sobrecargado. El constructor de la clase invoca la primera versión del operador `new` para crear los datos dinámicos de la clase `Matriz`. La llamada a `new` en `main` invoca a la segunda versión sobrecargada del operador `new` para crear una instancia dinámica de `Matriz`. La creación de la instancia implica al constructor, que invoca la primera versión del operador `new` para asignar espacio dinámico adicional, al que se accede por el puntero `Ptrdatos`. Para invocar al operador global `new` se debe utilizar la sintaxis `::new`.

21.10.2. Sobrecarga del operador `delete`

Se puede sobrecargar el operador `delete` en C++. En contraste con el operador `new`, se puede declarar sólo una versión sobrecargada por clase. La sintaxis general para declarar el operador `delete` sobrecargado es:

```
void operator delete(void *p, size_t tamaño)
```

Cuando se utiliza este segundo formato, el compilador pasará en `tamanyo` el número de bytes a liberar.

21.11. CONVERSIÓN DE DATOS Y OPERADORES DE CONVERSIÓN FORZADA DE TIPOS

Cuando en C++ una sentencia de asignación asigna un valor de un tipo estándar a una variable de otro tipo estándar, C++ convierte automáticamente el valor al mismo tipo que la variable receptora, haciendo que los dos tipos sean compatibles. Por ejemplo, las siguientes sentencias generan conversión de tipos numéricos:

```
long cuenta = 8;   // el valor 8 se convierte a un tipo long
double hora = 15;  // el valor 15 se convierte en un tipo double
int lado = 4.44;   // el valor double 4.44 se convierte en un int 4
```

C++ reconoce las conversiones de tipo, pero en algunas ocasiones se puede perder precisión. Por ejemplo, asignando `4.44` a la variable entera `lado` se asigna a dicha variable el valor 4 y se perderá la parte decimal 0,44.

Sin embargo, el lenguaje C++ no convierte automáticamente tipos que no son compatibles. Por ejemplo, la sentencia

```
int *p = 50;
```

produce un error, ya que en el lado izquierdo hay un tipo puntero, mientras que en el lado derecho hay un número. Los punteros y los enteros son conceptualmente diferentes. Por ejemplo, no se puede elevar al cuadrado un puntero. Sin embargo, cuando la conversión automática falla, se puede utilizar una conversión forzada de tipos (*cast*).

```
int *p =(int *) 50;      // p y (int *) 50 son punteros
```

El compilador funcionará de modo diferente según que la conversión sea entre tipos básicos, tipos básicos y objetos, o bien entre objetos de diferentes clases.

21.11.1. Conversión entre tipos básicos

Cuando se escribe una sentencia como

```
varentera = varfloat;
```

donde `varentera` es una variedad de tipo `int` y `varfloat` es una variable de tipo `float`, se supone que el compilador llama una rutina especial que convierte el valor de `varfloat`, en coma flotante, a un formato entero que se puede asignar a `varentera`. Existen naturalmente muchas conversiones `char` a `float`, `float` a `double`, etc. Cada conversión tiene su propia rutina, a la que se llama cuando los tipos de datos son diferentes a ambos lados. A esta conversión se la denomina *implícita*.

Con frecuencia, se necesita forzar al compilador para convertir un tipo a otro. Para realizar esta operación se utiliza el operador de *molde* (*cast*) o conversión forzada de tipos. Por ejemplo, para convertir `float` a `int` se puede expresar

```
varentera = int(varfloat);
```

El moldeado (*casting*) es una *conversión explícita* y utiliza las mismas rutinas incorporadas a la conversión implícita.

21.11.2. Conversión entre objetos y tipos básicos

En la conversión entre tipos definidos por el usuario y tipos básicos (incorporados) no se pueden utilizar las rutinas de conversión implícitas. En su lugar se deben escribir estas rutinas. Así, para convertir un tipo estándar o básico —por ejemplo, float— a un tipo definido por el usuario, por ejemplo `Distancia`, se utilizará un *constructor con un argumento*. Por el contrario, para convertir un tipo definido por el usuario a un tipo básico se ha de emplear un *operador de conversión*.

21.11.3. Funciones de conversión

Las *funciones de conversión* son funciones operador. El argumento en estas funciones es el argumento implícito `this`.

¿Cómo crear una función de conversión? Para convertir a un tipo *nombretipo* se utiliza una función de conversión con este formato:

```
operator nombretipo();
```

y se deben tener presentes los siguientes puntos:

- La función de conversión debe ser un método de la clase a convertir.
- La función de conversión no debe especificar un tipo de retorno.
- La función de conversión no debe tener argumentos.
- El nombre de la función debe ser el del tipo al que se quiere convertir el objeto.

Por ejemplo, una función para convertir a un tipo `double` tendrá este prototipo:

```
operator double();
```

Veamos una función de conversión diseñada para convertir clases diferentes: `Rect`, que representa la posición de un punto en el sistema de coordenadas rectangulares, y `Polar`, que representa la posición del mismo punto en el sistema de coordenadas polares.

Recordemos que las coordenadas de un punto en el plano se pueden representar por las coordenadas rectangulares `X` e `Y`, o bien por las coordenadas polares `Radio` y `Angulo.`

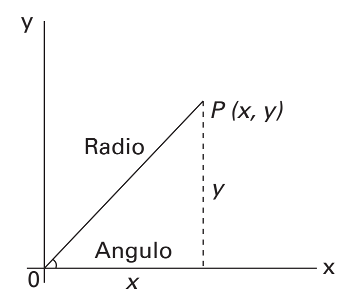


Figura 21.4. Coordenadas rectangulares de un punto.

Así pues, considerando las clases `Rec` y `Polar`, como son de estructura diferente, la sentencia de asignación

```
rect = polar;
```

implica una conversión que realizaremos mediante la función de conversión.

Veamos un programa `polarrec.cpp` que convierte coordenadas polares a rectangulares.

Recordemos las fórmulas a utilizar: *x = radio*coseno(ángulo)*
*y = radio*seno(ángulo)*

```
// polarrec.cpp
// convierte de polares a rectangulares

#include <iostream.h>

#include <math.h>

class Rect
{
   private:
      double xco;      // coordenada x
      double yco;      // coordenada y
   public:
```

```
      Rect()
         {xco = 0.0; yco = 0.0;}// constructor 0, sin argumentos

      Rect(double x, double y) // constructor 2, dos argumentos
         {xco = x; yco = y;}

      void visualizar()
         {cout << “ ( “ << xco << " , " << yco << " ) ";}
};

class Polar
{

   private:
      double radio;
      double angulo;
   public:
      Polar()                   // constructor sin argumentos
         {radio = 0.0; ángulo = 0.0}

      Polar(double r, double a) // constructor dos argumentos

         {radio = r; ángulo = a ;}

      void visualizar()
      {
         cout << " ( " << radio << " , " << ángulo << " ) ";
      }

      operator Rect()     // función de conversión
      {
         double x = radio * cos(ángulo); // calcular x
         double y = radio * sin(ángulo);       // calcular y
         return Rect(x, y);     // invoca al constructor de la clase Rect
      }

   };

   int main()
   {
      Rect rec;                  // Rect utiliza constructor 0
      Polar pol(100.0,0.785398); // Polar que utiliza constructor 2
      rec = pol;                  // convierte Polar a Rect
                           // utilizando función de conversión
                                // o bien rec = Rect(pol);

      cout << "\n polar="; pol.visualizar();  // polar original
      cout << "\n rectang=";rec.visualizar(); // rectangular
      return 0;                            // equivalente
}
```

La función `operator Rect()` transforma el objeto del cual es miembro en un objeto `Rect` y devuelve este objeto, que `main()` asigna a `Rec`.

La salida del programa para unos datos iniciales de *100,0* de radio y un ángulo de *45° (0,785398 radianes).*

```
polar = (100,00785398)
rectang = (70.710690,70.7106767)
```

21.11.4. Constructores de conversión

También existen constructores que permiten realizar conversiones entre tipos.

En el ejemplo visto anteriormente, la conversión de un objeto `Polar` en un objeto `Rect` se podría realizar definiendo un constructor de conversión en la clase `Rect`, de la siguiente forma:

```
class Rect {
      double xco;
      double yco;
public:
   Rect (Polar & p) (xco = p. rad() * cos (p.ang()); // rad() es una
                                                     // función miembro
                                                     // de polar que
                                                     // devuelve el
                                                     // radio
      yco = p. rad(); * sen (p. ang());              // ang() es una
                                                     // función miembro
                                                     // de polar que
                                                     // devuelve el
                                                     // ángulo
      }
// ...
};
int main() {
Rect Rec;
Polar Pol;
Rec = Pol;  // convierte Pol a Rec invocando al constructor de
            // conversión
// ...
}
```

21.12. MANIPULACIÓN DE SOBRECARGA DE OPERADORES

La sobrecarga de operadores es una de las operaciones fundamentales a utilizar en programación orientada a objetos. En los siguientes párrafos se enumeran las características más importantes a tener en cuenta en su manipulación.

1. La Tabla 21.1 recoge todos los operadores que se pueden sobrecargar (*los operadores predefinidos*).
2. Los operadores que no se pueden sobrecargar son:
   `.      .*      ::      ?:`
3. Los símbolos del preprocesador # y ## no se pueden sobrecargar.
4. Los operadores se pueden sobrecargar como funciones ordinarias, funciones amigas (`friend`) y funciones miembro.

5. La prioridad, asociatividad y el número de operandos asociados con los operadores predefinidos no se pueden cambiar al sobrecargarlos.
6. Un operador definido como una función miembro tiene un parámetro menos que si estuviera definido como una función ordinaria o como una función amiga.
7. Un operador definido como una función miembro se puede invocar utilizando la notación de función miembro o utilizando la notación prefijo, infijo o sufijo. Por ejemplo, considere la clase `T` que sobrecarga el operador binario más (+).

```
class T {
   // ...
public:
   // ...
   T operator+(T);
   // ...
};
```

Suponiendo que `a` y `b` son variables tipo `T`, entonces el operador `T::operator+` se puede invocar como:

```
a.operator+(b)
```

o bien como

```
a + b
```

La última invocación es exactamente el medio en que el operador más (+) que se invoca ha sido definida como `amiga` o como una función ordinaria. Como `amiga`

```
class T {
   // ...
public:
   friend T operator+(T,T);
   // ...
};
```

y como función ordinaria

```
T operator+(T,T);
```

La ambigüedad se puede evitar definiendo un operador por alguno de los tres métodos definidos anteriormente.

8. Los versiones sobrecargadas de los siguientes operadores se deben especificar como funciones miembro.

`=`	*asignación*
`()`	*llamada a función*
`[]`	*subíndice*
`->`	*desreferencia (indirección)*

9. En el caso de los operadores predefinidos `[]`, `*` y `->` se mantienen las siguientes equivalencias:

`p -> x`	`(*p).x`
`*(p+i)`	`p[i]`

`p` es un puntero a una clase con un miembro `x` e `i` es un entero. Si los operadores `[]`, `*` y `->` están sobrecargados, entonces se deben restablecer equivalencias explícitamente por el programador mediante definiciones apropiadas.

10. Las relaciones entre operadores sobrecargados no se establecen automáticamente. Por ejemplo, si ++ está sobrecargado en el tipo `complex` (complejo) para incrementar la parte real de valores complejos en 1,0, entonces

    ```
    a++
    ```
    donde a es de tipo complejo, no será equivalente a:

    ```
    a+= 1.0
    ```

    Tales relaciones existen *a priori* sólo para operadores predefinidos. Para establecer las relaciones anteriores entre los operadores ++ y -- para valores complejos, ambos operadores se deben sobrecargar apropiadamente.
11. Los operadores sobrecargados no pueden tener argumentos por defecto.

21.13. UNA APLICACIÓN DE SOBRECARGA DE OPERADORES

Consideremos un tipo `complex (complejo)` que representa operaciones típicas sobre números complejos (sumar, restar, multiplicar, dividir, igualdad) en el cual están sobrecargados los operadores `+, -, *, /` y ==.

Nota: Dados dos números complejos

```
z1 = a, ib   z2 = c, id
```

Las operaciones típicas son:

```
z1 + z2 = (a + c), i(b + d)
z1 - z2 = (a - c), i(b - d)
z1 * z2 = (ac - bd), i(bc + ad)
```

z1 / z2 = $\frac{(ac + bd)}{c^2 + d^2}, \frac{i(bc - ad)}{c^2 + d^2}$

La declaración de la clase `complex` se compila en el archivo *complex.h* y su definición o implementación de funciones miembro en el archivo `complex.cpp`

```
// complex.h

class complex {
   float r, i;
public:
   complex (float a = 0.0, float b = 0.0);
   float real();
   float imag();
   complex operator + (const complex& a);
   complex operator - (const complex& a);
   complex operator * (const complex& a);
   complex operator / (const complex& a);
   bool operator == (complex a);
} ;

   // archivo complex.cpp
   #include "complex.h"
   complex::complex(float a, float b)
```

```
{
   r = a-; i = b;
}
float complex::real()
{
   return r;
}

float complex::imag()
{
   return i;
}

complex complex::operator +(const complex& a)
{
   complex b(r + a.r, i + a.i);
   return b;
}

complex complex::operator -({const complex& a})
{
   complex b(r - a.r, i - a.i);
   return b;
}

complex complex::operator *(const complex& a)
{
   complex b(r * a.r - i*a.i, r * a.i + i * a.r);
   return b;
}

complex complex::operator /(const complex& a)
{
   complex b ;
   float denom = a.r * a.r + a.i * a.i;
   b.r = (r * a.r + i * a.i)/denom;
   b.i = (i * a.r-r * a.i)/denom;
   return b;
}

int complex::operator == (complex a)
{
   return (r == a.r) && (i == a.i);
}
```

Una vez definida la clase `complex`, se desea diseñar un programa que simule una calculadora aritmética que permita efectuar las operaciones aritméticas sobre números complejos leídos del teclado. El archivo `complejo.cpp` realiza esta simulación.

```
// archivo complejo.cpp

// calculadora para aritmética de complejos
```

```
#include <iostreeam.h>
#include “complex.h”

int main()
{
   float ar, ai, br, bi;
   char opr;
   complex r;
   for(;;) {
      cout << "Introduzca primer número complejo";
      cin >> ar; cin >> ai;
      cout << "introduzca operador: "; cin >> opr;
      cout << "introduzca segundo complejo: ";
      cin >> br; cin >> bi;
      switch (opr) {
         case '+':
            r = complex(ar, ai) + complex(br, bi);
            break;
         case '-':
            r = complex(ar, ai) - complex(br, bi);
            break;
         case '*':
            r = complex(ar, ai) * complex(br, bi);
            break;
         case '/':
            r = complex(ar, ai) / complex(br,bi);
            break;
         default:
            cerr << "operador no válido" << endl;
      }

      cout << "El resultado es-: ";
      cout << r.real() << “ ” << r.imag() << endl;
   }
}

// El programa se termina tecleando control-c
```

RESUMEN

- La sobrecarga de operadores representa nuevos significados. Por ejemplo, el significado de la expresión `a + b` depende de los tipos de las variables `a` y `b`. La expresión puede significar concatenación de cadenas, suma de números complejos o suma de enteros, dependiendo de que las variables sean de tipo cadena, complejo o entero.
- Todos los operadores C++ se pueden sobrecargar, con la excepción de:

  `.` `.*` `::` `?:`

  y los operadores del preprocesador # y ##.
- La palabra reservada `operator` se utiliza también para sobrecargar los operadores integrados C++. Al igual que un nombre de una función `printf()` puede tener una variedad de significados que depende de sus argumentos, así un operador, tal como `+`, puede tener significados adicionales.
- La sobrecarga de operadores utiliza normalmente funciones miembro o funciones amiga ya que ambas tienen privilegios de acceso. Cuando un operador se sobrecarga utilizando una función miembro, tiene una lista de argumentos vacíos, ya que el único argumento del operador es el argumento implícito. Cuando un operador binario se sobrecarga utilizando una función miembro, tiene como primer argumento la variable de clase pasada implícitamente y como segundo argumento el parámetro de la lista de argumentos.
- Un operador sobrecargado puede ser una función no miembro (normalmente una amiga, `friend`).

```
class x {
// ...
     friend x operator(const x &u,
const x &v);
};
```

o bien una función miembro

```
class complejo {
// ...
   complejo &operator + = (const com-
plejo &n);
};
```

- Los operadores sobrecargados, cuando se implementan como funciones miembro, pueden también utilizar el especificador `const`.

```
class complejo {
// ...
   complejo operator –() const;
};
```

- La sobrecarga de operadores se implementa por una función operador. Las funciones miembro tienen un argumento implícito, que es el puntero `this` oculto. La relación entre un operador y las correspondientes llamadas a funciones se resumen en

Operador	**Llamada a función**	
	Miembro	**No miembro**
`A <op> B`	`A.operator <op> B`	`Operator <op> (A, B)`
`<op> B`	`A.operator <op>()`	`Operator <op> (A)`
`A<op>`	`A.operator <op>(0)`	`Operator <op>(A, 0)`

LECTURAS RECOMENDADAS

- Joyanes Aguilar, Luis, *Programación Orientada a Objetos*, 2.ª edición, Madrid, McGraw-Hill, 1998.
- Lipman, Stanley B., y Lajoie, Josée, C++ Primer, 3.ª edición, Reading, Massachusetts, Addison-Wesley, 1998.
- Pohl, Yra, *Object-Oriented Programming using C++*, 2.ª edición, Reading, Massachusetts, Addison-Wesley, 1997.
- Joyanes, Luis, y Castán, Héctor, C++. *Iniciación y Referencia*, Madrid, McGraw-Hill, 1999.

EJERCICIOS

21.1. Escribir una clase `Racional` para números racionales. Un número racional es un número que puede ser representado como el cociente de dos enteros. Por ejemplo, 1/2, 3/4, 4/2, etc. Son todos números racionales. Sobrecargar los restantes de los siguientes operadores de modo que se apliquen al tipo `Racional :. ==, <, <=, >,>=,+, -, * y /`.

21.2. Definir una clase Complejo para describir números complejos. Un número complejo es un número cuyo formato es:

a + b * *i*

donde *a* y *b* son números de tipo `double` e *i* es un número que representa la cantidad $\sqrt{-1}$. Las variables *a* y *b* se denominan reales e imaginarias. Sobrecargar los siguientes operadores de modo que se apliquen correctamente al tipo `Complejo-: =, +, -, *, >> y <<`. Para añadir o restar dos números complejos, se suman o restan las dos variables miembro de tipo `double`. El producto de dos números complejos viene dado por la fórmula siguiente:

*(a+b*i)*(c+d*i)==(a*c-b*d)+(a*d+b*c)*i*

21.3. Escribir una clase `Hora_reloj` que permita expresar la hora del día en horas y minutos, utilizando un reloj de 24 horas. La clase debe sobrecargar a los operadores `++` y `--` que incrementen o decrementen las horas y minutos del reloj.

21.4. Describir una clase `Punto` que defina la posición de un punto en un plano cartesiano de dos dimensiones. Diseñar e implementar una clase `Punto` que incluya la función operador (`-`) para encontrar la distancia entre dos puntos.

21.5. Crear un operador función que permita sumar, restar y multiplicar dos matrices.

21.6. Definir los operadores necesarios para realizar operaciones con cadenas tales como sumar (concatenar), comparar o extraer cadenas.

21.7. El *producto escalar* de dos vectores de igual longitud (u_1, u_2, ..., u_n) y (v_1, v_2, ... v_n) se puede definir como la suma

$$\sum_{k=1}^{n} u_k \, v_k$$

Completar la clase `Vector` con un operador que calcule el producto escalar de dos vectores.

21.8. Escribir una clase `Vector` que permita sumar, restar, multiplicar y dividir vectores.

CAPÍTULO 22

Excepciones

Contenido

INTRODUCCIÓN

Uno de los problemas más importantes en el desarrollo de software es la gestión de condiciones de error. No importa lo bueno que sea el software y la calidad del mismo, siempre aparecerán errores por múltiples razones (errores de programación, errores imprevistos de los sistemas operativos, agotamiento de recursos, etc.). *Excepciones* son, normalmente, condiciones de errores imprevistos. Estas condiciones suelen terminar el programa del usuario con un mensaje de error proporcionado por el sistema. Ejemplos típicos son: agotamiento de memoria, errores de rango en intervalos, división por cero, etcétera. El rango y definición de estos errores, así como el modo en que se manejan los errores, pueden ser definidos por el programador. *El manejo de excepciones* es el mecanismo previsto por C++ para el tratamiento de excepciones. Normalmente el sistema aborta la ejecución del programa cuando se produce una excepción y C++ permite al programador intentar la recuperación de estas condiciones y continuar la ejecución del programa.

CONCEPTOS CLAVE

- Captura de excepciones.
- `catch`.
- El bloque `try`.
- Especificación de excepciones.
- Excepción.
- Excepciones estándar.
- Lanzamiento de excepciones.
- Levantar una excepción.
- Manejador de excepciones.
- Manejo de excepciones.
- `terminate()` y `unexpected`.
- `throw`.

22.1. CONDICIONES DE ERROR EN PROGRAMAS

La escritura de código fuente y el diseño correcto de clases y funciones es una tarea difícil y delicada, por ello es necesario manejar los errores que se produzcan con la mayor eficiencia. La mayoría de los diseñadores y programadores se enfrentan a dos importantes problemas en el manejo de errores:

1. ¿Qué tipo de problemas se pueden esperar cuando los clientes hacen mal uso de clases, funciones y programas en general?
2. ¿Qué acciones se deben tomar una vez que se detectan estos problemas?

El manejo de errores es una etapa importante en el diseño de programas ya que no siempre se puede asegurar que las aplicaciones utilizarán objetos o llamadas a funciones correctamente. En lugar de añadir conceptos aislados de manejo de errores al código del programa, es deseable construir un mecanismo de manejo de errores como una parte integral del proceso de diseño.

Para ayudar a la *portabilidad* y *diseño* de bibliotecas, C++ incluye un *mecanismo de manejo de excepciones* que está soportado por el lenguaje. En este capítulo se trata de explorar el manejo de excepciones y cómo utilizarlo en su diseño. Para ello se examinarán: detección de errores, manejo de errores, gestión de recursos y especificaciones de excepciones.

22.1.1. ¿Por qué considerar las condiciones de error?

La *captura* (*catch*) o *localización* de errores inadecuados ha sido siempre un problema en el software. Parte del problema es que la captura de errores es una labor muy importante en el trabajo de un programador. También, otros factores a tener en cuenta son: ¿dónde deben ser capturados los errores? o ¿cómo pueden ser manejados? Una vez que se encuentra un error, un programa debe tener varias opciones: terminar inmediatamente; ignorar el error con la esperanza de que no suceda nada «desastroso»; o bien puede establecer un indicador o señal de error, el cual (presumiblemente) se comprobará por otras sentencias del programa. En esta última opción, cada vez que se llama una función específica, el llamador debe comprobar el valor de retorno de la función, y si se detecta un error, se debe determinar un medio de recuperar el error o de terminar el programa.

En la práctica, los programadores no son consistentes en estas tareas, debido en parte a que exige una gran cantidad de trabajo y también debido a que las sentencias de verificación de errores a veces oscurecen la comprensión del resto del código. También es difícil recordar cómo capturar («atrapar») cada condición posible de error cada vez que se llama una función determinada. Con frecuencia, el programador debe hacer esfuerzos excepcionales para invocar a los destructores, liberar memoria y cerrar archivos de datos antes de detener un programa.

La solución a tales problemas en C++ es llamar a mecanismos del lenguaje que soportan manejo de errores que eviten la realización de códigos de manejo de errores complejos y artificiales en cada programa. En C++ cuando se genera una excepción, el error no se puede ignorar o el programa terminará. Si el código de tratamiento del error está en lugar de un tipo de error específico, el programa tiene la opción de recuperación del error y continuar la ejecución. Este enfoque ayuda a asegurar que no se deslice ningún error que produzca consecuencias fatales y origine que un programa se comporte erráticamente.

Un programa «lanza» (*throws*) una excepción en el punto en que primero se detecta el error. Cuando esto sucede, un programa C++ busca automáticamente en un bloque de código llamado *manejador de excepciones*, que responde a la excepción de un modo apropiado. Esta respuesta se llama «capturar o atrapar una excepción» (*catching an exception*). Si no se puede encontrar un manejador de excepciones, el programa, simplemente, termina.

22.2. EL TRATAMIENTO DE LOS CÓDIGOS DE ERROR

Los desarrolladores de software han estado utilizando, durante mucho tiempo, códigos para indicar condiciones de error. Normalmente, los procedimientos devuelven un código para indicar sus resultados. Por

ejemplo, un procedimiento `AbrirArchivo` puede devolver 0 (cero) para indicar un fallo. Otros procedimientos pueden devolver (–1) para indicar un fallo y (0) para indicar éxito. Los sistemas operativos y las bibliotecas de software documentan todos los códigos de error posibles que pueden ser devueltos por un procedimiento. Los programadores que utilizan tales procedimientos deben comprobar cuidadosamente el código devuelto y las cosas a realizar en función de los resultados. Esto puede producir más serios problemas en el código posterior. Cuando una aplicación termina anormalmente pueden suceder cosas anómalas tales cómo: archivos que se abrieron no se cierran, conexiones de redes no se cierran, datos no se escriben en disco. *Una aplicación bien diseñada e implementada no debe permitir que esto suceda.*

Los errores no se deben propagar innecesariamente. Si los errores se detectan y se manejan tan pronto como suceden, se evitan daños mayores en código posterior. Cuando los errores se propagan a diferentes partes del código, será mucho más difícil *trazar* la causa real del problema debido a que los síntomas del problema pueden no indicar la causa real.

Cuando se detecta un error en una aplicación, se debe poder fijar la causa del problema y volver a intentar (procesar) la operación que produjo el error. Por ejemplo, si se ha producido un error porque el índice de un array se había fijado más alto que el final de dicho array, es fácil localizar la causa de error. En otras situaciones, la aplicación puede no ser capaz de fijar la condición de error. Una salida elegante puede ser la única salida en tales situaciones. El programador debe tener la libertad de tomar la decisión correcta.

Cuando un procedimiento encuentra una condición de error y vuelve al llamador, se debe esperar que el procedimiento ejecute las operaciones de «limpieza» necesarias antes de retornar al llamador. El código que detecta y maneja la condición de error debe incorporar código extra para esta operación de limpieza. En otras palabras, no es una tarea fácil y es más difícil cuando hay múltiples puntos de salida del procedimiento. Si existen múltiples posiciones en el código en las que se deben verificar las condiciones de error, el procedimiento se volverá complicado y enrevesado.

En ciertos casos, un procedimiento puede no tener información suficiente para manejar una condición de error; en estos casos puede ser más seguro propagar las condiciones de error a un procedimiento exterior en el que se pueda manejar. Los procedimientos que devuelven códigos de error sencillos pueden no ser capaces de cumplir estos retos.

Los códigos de error devueltos de procedimientos no transmiten mucha información al procedimiento llamador. Es normalmente un número que indica la causa del fallo. Sin embargo, en muchas situaciones puede ser muy útil si existe más información disponible sobre causa del fallo en el llamador. Esto ayudará a fijar la condición de error (si es posible). Un código de error sencillo no puede cumplir este objetivo.

En resumen*, las alternativas típicas para el manejo de errores* en programas son:

1. Terminar el programa.
2. Devolver un valor que representa un error.
3. Devolver un valor legal, pero establecer un indicador (señal) de error global.
4. Llamar a una función de error proporcionada por el usuario.

Cualquiera de los métodos tiene inconvenientes y deficiencias, en ocasiones graves. Por esta causa es necesario buscar nuevos métodos o esquemas. Uno de los esquemas más populares que ha sido comprobado y está soportado en muchos lenguajes (C++, Ada, Java...) es el principio de «levantamiento» o «alzamiento» (*raising*) de una excepción. En este principio se fundamenta el mecanismo de manejo de excepciones de C++ que ya se ha citado anteriormente y que trataremos ahora.

22.3. MANEJO DE EXCEPCIONES EN C++

Una excepción se «levanta» (*raise*) en caso de un error e indica una condición anormal que no se debe encontrar durante la ejecución normal de código. Una excepción indica una necesidad urgente de tomar una acción reparadora (de remedio).

La palabra *excepción* indica aquí una excepción software. No se debe confundir con una excepción hardware. Una excepción software se produce por una parte de código escrita por un programador. No tiene nada que ver con las excepciones hardware. Una excepción software se inicia por alguna parte del código que encuentra una condición anormal.

Una excepción es un error de programa que ocurre durante la ejecución. Si ocurre una excepción y está activo un segmento de código denominado *manejador de excepción* para esa excepción, entonces el flujo de control se transfiere al manejador. Si en lugar de ello ocurre una excepción y no existe un manejador para la excepción, el programa se termina.

Una excepción se puede levantar cuando «el contrato» entre el *llamador* y el *llamado* se violan. Por ejemplo, si se intenta acceder a un elemento que está fuera del rango válido de un array se produce una violación del contrato entre la función que controla los índices (`operator[]` en C++) y el llamador que utiliza el array. La función de índices garantiza que devuelve el elemento de la función especificada si el índice que se le ha pasado es válido. Pero si el índice no es válido, la función de índice debe indicar la condición de error. Siempre que se produzca tal violación del contrato se debe levantar (*alzar*) una excepción.

Una vez que se levanta una excepción, ésta no desaparece aunque el programador lo ignore (una excepción no se puede ignorar ni suprimir). Una condición de excepción se debe *reconocer* y *manejar*. Una excepción no manejada se propagará dinámicamente hasta alcanzar el nivel más alto de la función (`main` en C++). Si también falla el nivel de función más alto, la aplicación se termina sin opción posible.

Los lenguajes que soportan el mecanismo de excepciones tienden a mantener el código de excepciones bastante independiente del código normal. Ello es debido a que el mecanismo de manejo de errores —no importa lo bueno que sea—, es inútil si se degrada el rendimiento (prestaciones) del código normal.

Precaución

El manejo de errores usando excepciones no evita errores, sólo permite la detección y posible recuperación de los mismos. La evitación de errores se puede conseguir utilizando aserciones (`assert`).

En general, el mecanismo de excepciones en C++ (y en la mayoría de los lenguajes) permite:

1. Detección de errores enérgica y posible recuperación.
2. Limpieza y salida elegante en caso de errores no manejados.
3. Propagación sistemática de errores en una cadena de llamadas dinámicas.

Inconvenientes

La inclusión de un mecanismo de gestión y manejo de excepciones al código normal para manejar errores no es gratuito. Se debe añadir nuevo código a los procedimientos que gestionan las excepciones. El código de gestión de excepciones entremezclado con el código normal, oscurece la lógica del procedimiento original. Por último, añadir excepciones a una aplicación incrementa también el tamaño del código ejecutable.

22.4. EL MECANISMO DE MANEJO DE EXCEPCIONES

En C++, una excepción es un objeto; es decir, un valor, de un tipo fundamental incorporado en el lenguaje o de un tipo definido por el usuario (normalmente una clave). En la práctica, una excepción es un

objeto que se pasa desde el área de código donde surge un problema al área de código que va a manipular el programa. Cuando ocurre una excepción se dice que se *lanza* (en inglés, *throw*) en una función donde se detecta un error o un problema, y cuando se manipula una excepción se dice que se ha capturado (*caught*, en inglés) en otra función donde se puede tratar o procesar (en inglés, *try*) el suceso. Si no se captura la excepción se tiene previsto un comportamiento bien definido.

En realidad el manejo de excepciones permite que partes de un programa desarrolladas independientemente se comuniquen para manejar problemas que surgen durante la ejecución del programa. Una parte de un programa puede detectar un problema que esa parte del programa no puede resolver. Es decir, la parte que detecta el problema puede pasar el problema junto a otra parte que se prepara para manejar aquello que está mal.

En esencia, *las excepciones nos permiten separar la detección del problema de la resolución del problema. La parte del programa que detecta un problema no necesita conocer cómo se debe tratar dicho problema.*

El manejo de errores y otros comportamientos anómalos pueden ser una de las partes más difíciles del diseño de cualquier sistema. Los sistemas interactivos más usuales, hoy día, en el mundo de las comunicaciones tales como los conmutadores (*switches*) y los encaminadores (*routers*) pueden dedicar hasta un noventa por ciento de su código a detección de errores y manipulación de los mismos. Con la proliferación de aplicaciones basadas en web que se ejecutan indefinidamente, la tarea de manipulación de errores se ha convertido cada vez más importante para la mayoría de los programadores.

Las excepciones son anomalías en tiempo de ejecución tales como agotamiento de la memoria o encontrarse con una entrada no prevista. Las excepciones existen fuera del funcionamiento normal de un programa y requiere de manipulación inmediata por parte del programa. En sistemas bien diseñados, las excepciones representan un subconjunto de la manipulación de errores. *La parte del programa que detecta el problema necesita un medio para transfeir el control a la parte del programa que puede manipular el problema.* La parte de detección necesita también poder indicar qué tipo de problema ha ocurrido y debe tener que proporcionar información adicional.

Las excepciones soportan el tipo de comunicación entre las partes de un programa que detectan el error y las partes que manipulan el error. Las excepciones se procesan de acuerdo a los siguientes conceptos:

- Si ocurre una situación anormal en una función, ésta se comunica con el *llamador* con una sentencia especial. Se cambia entonces del flujo normal de datos al manejo de excepciones. Esta manipulación de excepciones deja todos los bloques o funciones llamados hasta que se encuentra un bloque en el que se puede manipular una excepción.
- Cuando se ejecutan las sentencias, podemos definir lo que sucede si una situación de excepción ocurre. Un bloque individual de sentencias puede entonces determinar cómo se trata esa situación.

22.4.1. Claves de excepciones

C++ adopta un enfoque orientado a objetos para manipular excepciones:

- Las excepciones se consideran como objetos. Si ocurre una excepción, se crea un objeto correspondiente que describe la excepción.
- Las claves de excepción definen las propiedades de este objeto. Las propiedades de una excepción se pueden definir como atributos (por ejemplo, un índice incorrecto o un mensaje explicatorio). Las operaciones se pueden utilizar para consultar información fuera de una excepción. Para tipos diferentes de excepciones existe también la capacidad de proporcionar claves correspondientes diferentes. Cada clave de excepción describe un tipo particular de excepción.

Cuando ocurre una excepción, se crea un objeto de la correspondiente clave de excepción. Al igual que cualquier otro objeto, éstos poseen miembros que describen las excepciones y las condiciones en las que ocurren. Ellos son, por consiguiente, parámetros de la excepción.

Se pueden formar jerarquías de claves de excepción utilizando licencia. Por ejemplo, los errores matemáticos se pueden dividir en claves especiales de «división por cero», «raíz cuadrada de números negativos», etc.; de este modo, un programa de aplicación puede manejar errores matemáticos, en general, o divisiones por cero, en particular.

22.4.2. Partes de la manipulación de excepciones

En C++, la manipulación de excepciones implica:

- *expresiones* **`throw`**. Son utilizadas por la parte de detección de errores para indicar que se ha encontrado un error que no se puede manipular. Se dice que una sentencia `throw` levanta (*raise*) una condición excepcional.

  `throw` se utiliza para «lanzar» un objeto excepción en un área de programa; de este modo se cambia desde el flujo de datos normal al manejo de excepciones.
- *bloques* **`try`**, que utilizan la parte de manejo de errores para tratar una excepción. Un bloque `try` comienza con la palabra reservada `try` y termina con una o más cláusulas `catch`. El bloque `try` indica un ámbito en el cual se interceptan las excepciones y se manipulan utilizando `catch`. Las excepciones lanzadas desde el código ejecutado en el interior de un bloque `try` se manipulan normalmente por una de las cláusulas `catch`. Puesto que ellas «manipulan» la excepción, las cláusulas `catch` se conocen como *manejadores* (*handlers*).

  `catch` se utiliza para «capturar» un objeto excepción y para reaccionar a la situación de excepción. Se utiliza para implementar lo que sucede cuando el flujo de datos no funciona.
- Un conjunto de **claves de excepciones** definidas en la biblioteca que son utilizadas para parar la información acerca de un error entre un `throw` y una `catch` asociada.

22.5. EL MECANISMO DE MANEJO DE EXCEPCIONES

El modelo de un mecanismo de excepciones consta, fundamentalmente, de tres nuevas palabras reservada `try, throw` y `catch`.

- `try`, un bloque para detectar excepciones;
- `catch`, un manejador para capturar excepciones de los bloques `try`;
- `throw`, una expresión para levantar (*raise*) excepciones.

Los pasos del modelo son:

1. Primero, un programador «intentará» (*try*) una operación para anticipar errores.
2. Cuando un procedimiento encuentra un error, se «lanza» (*throws*) una excepción. El lanzamiento (*throwing*) de una excepción es el acto de levantar una excepción.
3. Por último, alguien interesado en una condición de error (para limpieza y/o recuperación) anticipará el error y «capturará» (*catch*) la excepción que se ha lanzado.

El mecanismo de excepciones se completa con:

- Una función `terminate()` que atrapa las excepciones no capturadas.
- Especificaciones de excepciones que dictamina cuales excepciones, si existen, puede lanzar una función.
- Una función `unexpected()` que atrapa las violaciones de especificaciones de excepciones.

22.5.1. El modelo de manejo de excepciones

La filosofía que subyace en el modelo de manejo de excepciones es simple. El código que trata con un problema no es el mismo código que lo detecta. Este tipo de separación construye un cortafuegos (*firewall*) —similar a los establecidos en Internet— entre aplicaciones y bibliotecas de clases (Figura 22.1).

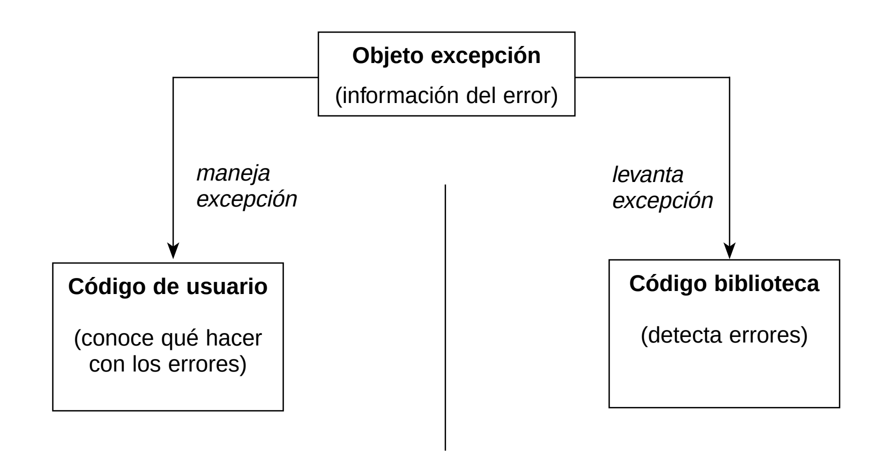


Figura 22.1. Manejo de excepciones construye un «cortafuegos».

Cuando una excepción se encuentra en un programa C++, la parte del programa que detecta la excepción puede comunicar que la expresión ha ocurrido levantando, o lanzando (*throwing*) una excepción.

De hecho, cuando el código de usuario llama a una función incorrectamente o utiliza un objeto de una clase inadecuadamente, la biblioteca de la clase crea un objeto excepción que contiene información sobre lo que era incorrecto. La biblioteca levanta (*raise*) una excepción, una acción que hace el objeto excepción disponible al código de usuario a través de un *manejador de excepciones*. El código de usuario que maneja la excepción puede decidir qué hacer con el objeto excepción. Este enfoque ofrece diversas ventajas. Los programas se hacen más legibles, dado que el código de manejo de errores es independiente del código que detecta los errores. Un estilo regular de manejo de errores es consistente a través de bibliotecas diferentes. Los usuarios han de tener también una regla acerca de qué hacer con excepciones.

El cortafuegos que actúa de puente entre una biblioteca de clases y una aplicación debe hacer varias cosas para gestionar debidamente el flujo de excepciones, reservando y liberando memoria de modo dinámico.

Una de las razones más significativas para utilizar excepciones es que las aplicaciones no pueden ignorarlas. Cuando el mecanismo de excepciones levanta una excepción, alguien debe tratarla. En caso contrario, la excepción es «no capturada» (*uncaught*) y el programa termina por omisión. Este poderoso concepto es el corazón del manejo de excepciones y fuerza a las aplicaciones a manejar excepciones en lugar de ignorarlas.

El mecanismo de excepciones de C++ sigue un *modelo de terminación*. Esto implica que nunca vuelve al punto en que se levanta una excepción. Las excepciones no son como manejadores de interrupciones que bifurcan a una rutina de servicio antes de volver al *spot* de interrupción. Esta técnica (llamada *resumption*) tiene un alto tiempo suplementario y es propensa a bucles infinitos y es más complicado de implementar que terminación. Diseñado para manejar sólo excepciones síncronas con un solo hilo de control, el mecanismo de excepciones implementa un camino alternativo de una sola vía en diferentes sitios de su programa.

Ejemplo 22.1

```
void f()
{
   // código que produce una excepción que se lanza
   // ...
   throw   i;
   //   ...
}

main()
{
   try {
      f(); //Llamada a f, preparada para cualquier error
      // Código normal aquí
   }
   catch(...)
   {
      // capturar cualquier excepción lanzada por f()
      // hacer algo
   }
   // Resto código normal de main()
}
```

`f()` es un simple procedimiento. Cuando se encuentra una condición de error, se lanza una excepción. Esto se consigue mediante la sentencia

```
throw i;
```

El operando de la expresión `throw` es un objeto. Normalmente los objetos con información sobre el error se lanzan.

En el programa `main()`, la llamada a `f()` se encierra en un bloque `try`. El código es un bloque `try`

```
try {
   // ...
}
```

En otras palabras, es un bloque de código encerrado dentro de una sentencia `try`. Un bloque `try` indica al compilador la posibilidad de una excepción.

Un bloque `catch()` captura una excepción del tipo indicado. En el ejemplo anterior

```
catch(...)
```

indica que captura excepciones de todo tipo. Los puntos suspensivos (...) significan cualquier argumento. Una expresión `catch` es comparable a un procedimiento con un argumento.

El código C++ puede levantar una excepción en un bloque `try` utilizando la expresión `throw`. La excepción se maneja invocando un manejador apropiado seleccionado de una lista de manejadores que se encuentran al final del bloque `try` del manejador.

22.5.2. Diseño de excepciones

La palabra reservada `try` designa un bloque *try*, que es un área de su programa que detecta excepciones. En el interior de bloques `try`, normalmente se llaman a funciones que pueden levantar o *lanzar* excep-

ciones. La palabra reservada `catch` designa un manejador de capturas con una signatura que representa un tipo de excepción. Los manejadores de captura siguen inmediatamente a bloques `try` o a otro manejador `catch` con una signatura diferente.

Se debe establecer un bloque `try` y un manejador de capturas (`catch`) para capturar excepciones lanzadas. En caso contrario, la excepción no es capturada y su programa termina por omisión. Los bloques `try` son importantes ya que sus manejadores de captura asociados determinan cuál es la parte de su programa que maneja una excepción específica. El código que está dentro del manejador de capturas (`catch`) es donde se decide lo que se hace con la excepción lanzada.

22.5.3. Bloques `try`

Un *bloque* `try` debe encerrar las sentencias que pueden lanzar excepciones y comienza con la palabra reservada `try` seguida por una secuencia de sentencias de programa encerradas entre llaves. A continuación del bloque `try` hay una lista de manejadores llamados *cláusulas* `catch`. Al menos un manejador `catch` debe aparecer inmediatamente después de un bloque `try` para manejar excepciones lanzadas; en caso contrario, el compilador genera errores. Cuando un tipo de excepción lanzada coincide con la signatura de un manejador `catch`, el control se reanuda dentro del bloque del manejador `catch`. Si ninguna excepción se lanza desde un bloque `try`, el control «salta» al manejador `catch` y prosigue a la sentencia siguiente.

La sintaxis del bloque `try` es

1
```
try {
    código del bloque try
}
catch (signatura) {
    código del bloque catch
}
```

2
```
try
sentencia compuesta
lista de manejadores
```

También se puede anidar bloques `try`.

```
void sub(int n)
{
   try {
      ...                        // bloque try externo
      try {                      // bloque interno try
         ...
         if (n==1)
            return ;
      }
      catch (signatura1) {...}   // manejador catch interno
   }
   catch (signatura2) {...}      // manejador catch externo
   ...
}
```

Una excepción lanzada en el bloque interior `try` ejecuta el manejador `catch` con *`signatura1`* si coincide el tipo de excepción. El manejador `catch` con *`signatura2`* maneja excepciones lanzadas desde el bloque `try` exterior si el tipo de la excepción coincide. El manejador externo de `catch` también captura excepciones lanzadas desde el bloque interior `try` si el tipo de excepción coincide con *signatura2* pero no con *signatura1*. Si los tipos de excepción no coinciden con ninguna signatura, la excepción se propaga al llamador de `sub()`.

NORMAS

```
try {
   sentencias
}
catch (parámetro) {
   sentencias
}
catch (parámetros) {
   sentencias
}
etc.
```

1. Cuando una excepción se produce en sentencias después de `try`, hay un salto al primer manejador (la parte `catch`) cuyo parámetro coincida con el tipo de excepción.
2. Cuando las sentencias en el manejador se han ejecutado, se termina el bloque `try` y la ejecución prosigue en la sentencia siguiente. No se produce nunca un salto hacia atrás al lugar en que ocurrió la interrupción.
3. Si no hay manejadores para tratar con una excepción, se aborta el bloque `try` y la excepción se activa.

El bloque `try` es el contexto para decidir qué manejadores se invocan en una excepción levantada. El orden en que los manejadores se definen, determina el orden de llamada. Una excepción sólo puede lanzarse después de que la ejecución de un programa se ha introducido en un bloque `try`.

Precaución

Se puede transformar el control fuera de bloques *try* con una sentencia `goto`, `return`, `break` o `continue`, pero no se puede transferir el control en bloques *try* con estas palabras reservadas.

Los bloques *try* sirven para muchos propósitos útiles.

Ejemplo 22.2

La función `calcu_media()` calcula una media de arrays de tipo double con una función plantilla `avg()` (cálculo de la media aritmética). En el caso de ser llamada incorrectamente, `avg()` lanza excepciones de cadena de caracteres.

```
double calcu_media(unsigned int long){
const unsigned int max = 6;
double b[max] = {1.2, 2.2, 3.3, 4.4, 5.5, 6.6};
if (lon > max) {
   cerr << "calcu_media: uso de longitud por defecto de"
        << max << endl;
   lon = max;
}
try {
   return avg(b, lon);       // cálculo media
}
```

```
catch (char *msg) {
   cerr << msg << endl;
   cerr << calcu_media: uso de longitud por defecto de"
         << max << endl;
  return avg(b, max);
   }
}
```

La función declara un array de 6 tipos `double` y llama a `avg()` con el nombre del array (`b`) y un argumento de longitud (`lon`). Un bloque `try` circunda a `avg()` para capturar excepciones de cadena de caracteres. Obsérvese que `calcu_media` asegura que `lon` no es nunca mayor que `max`.

Si `avg` lanza una excepción de cadena de caracteres, el manejador de *catch* recupera volviendo a llamar `avg()` con un valor por defecto (la longitud del array `b`). Esta técnica permite a `calcu_media` llamar a `avg()` una segunda vez con un valor correcto.

22.5.4. Lanzamiento de excepciones

La sentencia `throw` levanta una excepción. Cuando se encuentra una excepción en un programa C++, la parte del programa que detecta la excepción puede comunicar que la excepción ha ocurrido por levantamiento o *lanzamiento* de una excepción. El formato de `throw` es:

1. `throw` *`expresión`*
2. `throw`

El bloque más interno de `try` en el que se levanta una excepción se utiliza para seleccionar la sentencia `catch` que procesa la excepción. La sentencia `throw` sin argumento se puede utilizar dentro de un `catch` para *relanzar* la excepción actual. Normalmente se utiliza cuando se desea que un segundo manejador sea llamado desde el primer manejador para procesar posteriormente la excepción.

Ejemplo 22.3

```
// archivo throw1.cpp
void demo()
{
   int i;
   // lanzamiento de una excepción
   i = -16;
   throw i;
}
int main()
{
   try {
      demo()
   }
   catch(int n)
      cerr << "excepción capturada \n" << n << endl; }
}
```

El valor entero lanzado por `throw` persiste hasta que se sale del manejador `catch(int n)`. Este valor está disponible para usar con el manejador como su argumento.

22.5.5. Captura de una excepción: `catch`

Un manejador de excepciones C++ es una cláusula de captura (*catch*). Cuando una excepción se lanza desde sentencias dentro de un bloque *try*, la lista de cláusulas *catch* que siguen al bloque *try* se buscan para encontrar una cláusula *catch* que pueda manejar la excepción.

Una cláusula *catch* consta de tres puntos: la palabra reservada `catch`, la declaración de un solo tipo o un simple objeto dentro de paréntesis (referenciado como *declaración de excepciones*), y un conjunto de sentencias dentro de una sentencia compuesta. Si la cláusula *catch* se selecciona para manejar una excepción, se ejecuta la sentencia compuesta.

Formatos

1.
```
catch (signatura) {   captura signatura correspondiente excepción
    código del bloque catch
}
```

2.
```
catch (...) {                    captura cualquier excepción
    código del bloque catch
}
```

La especificación 1 del manejador `catch` se asemeja a una definición de función.

Tipo de excepción a procesar — *Nombre de la información pasada al manejador*

```
catch (TipoParámetro   NombreParámetro)  {
    CuerpoManejador
}
```

Lista de sentencias que procesan la excepción

El manejador de excepciones consta de la palabra reservada `catch` y de las sentencias que le siguen (*código del bloque* o *cuerpo Manejador*). Al contrario que el bloque `try`, el bloque `catch` sólo se ejecuta bajo circunstancias especiales. La signatura es una declaración de argumentos. Puede constar de un tipo, un tipo seguido por un nombre de argumento o una sintaxis especial:

```
catch (tipo)
catch (tipo nombre_argumento)
catch (...)
```

Cada una de las dos primeras sentencias crea un manejador de excepciones que busca tipos coincidentes. Al igual que con funciones sobrecargadas, el manejador de excepciones se activa sólo si el argumento que se pasa (o *se lanza*, «thrown») se corresponde con la declaración del argumento.

La tercera sentencia es una sintaxis especial que significa «cualquier excepción». Se puede utilizar esta sintaxis para escribir manejadores de excepciones por defecto que capturan todas las excepciones no capturadas ya.

La sintaxis completa de `try` y `catch` permite escribir cualquier número de manejadores de excepciones al mismo nivel.

```
try {
   sentencias
}
catch (parámetro1) {
   sentencias
}
catch (parámetro2) {
   sentencias
}
...
```

Los puntos suspensivos (...) significa que puede tener cualquier número de bloques `catch` a continuación del bloque `try`.

Funcionamiento

Cuando ocurre (se lanza) una excepción en una sentencia durante la ejecución del bloque `try`, el programa comprueba, por orden, cada bloque `catch` hasta que encuentra un manejador (la parte `catch`) cuyo parámetro coincide con el tipo de excepción. Tan pronto como se encuentra una coincidencia, se ejecutan las sentencias del bloque `catch`; cuando se han ejecutado las sentencias del manejador, se termina el bloque `try` y prosigue la ejecución con la siguiente sentencia. Es decir, si el programa no ha terminado, la ejecución se reanuda normalmente después del final de todos los bloques `catch` de la función actual.

Si no existen manejadores para tratar con una excepción, el bloque `try` se aborta y la excepción no es capturada. Cada manejador se introduce con la palabra reservada `catch` y tiene un único parámetro. Si no ocurre ninguna excepción, las sentencias se ejecutan de modo normal y ninguno de los manejadores será invocado.

Imaginemos la siguiente función:

```
int f(int x)
{
   int a = fa(x);
   int b = fb(x);
   return a / b;
}
```

Las funciones `fa` y `fb` suponemos que son dos funciones arbitrarias. Si la función `fb` devuelve un valor de 0, se tendrá un problema. Si no se hace nada, se realizará una división por cero, lo que proporcionará un resultado imprevisto y no definido (infinito; la mayoría de los compiladores de C++ no admiten la división por cero). Se trata de tomar el control de la excepción levantada (división por cero) y tratar de hacer alguna acción adecuada para permitir que continúe el programa. Una solución podía ser ésta:

```
int f(int x)
{
   int a = fa(x);
   int b = fb(x);
   if (b == 0)
      return INT_MAX;
   return a / b;
}
```

La constante `INT_MAX` se define en el archivo estándar `limits.h` e indica el número mayor que se puede almacenar en una variable de tipo `int`. Si, por ejemplo, el tipo `int` se representa con 16 bits, `INT_MAX` será igual a 32.767. Éste sería el número entero mayor que se devolvería en el caso de que `b` fuera igual a 0, cosa que naturalmente es absurda.

La solución pasa por escribir

```
if (b == 0)
{
   cout << "Desbordamiento en la función f";
   abort();
}
```

pero entonces el programa se detiene y puede que esto no sea necesario. El resultado de la función `f` puede no ser tan importante, después de todo. Si la función llamada `f` ha encontrado que se ha producido un error, puede proseguir su ejecución. En resumen, se puede escribir la nueva versión de la función `f` que genera una excepción cuando `b` se convierte en `0`.

```
int f(int x)
{
   int a = fa(x);
   int b = fb(x);
   if(b == 0)
      throw error_desbordamiento ("Error en f");
   return a/b;
}
```

Ejemplo

```
void g()
{
   int i;
   while (i)
      try {
         cout << "?";
         if (not cin >> i))
            break;

         int r = f(i);
         cout << "El resultado era " << r << endl;
      }
      catch(error_desbordamiento)
         cout << "No se puede dar resultado" << endl;
      }
}
```

La función `g` realiza la lectura de enteros del teclado. La entrada se termina cuando aparece la combinación de caracteres final de archivo (`Ctrl-Z` o bien `Ctrl-D`). Por cada número leído, se llama a la función `f` y el resultado de `f` se escribe en el terminal. Si se lee un número tal que `b` en la función `f` se hace igual a cero, se ejecuta la siguiente sentencia:

```
throw error_desbordamiento ("Error en f");
```

Dado que la excepción que ahora se produce no reside en un bloque `try`, no se tratará en la función `f`. En su lugar, `f` se interrumpirá y la excepción se envía más tarde a la función `g`, que es llamada en `f`. En `g` existe entonces una excepción la del punto en que fue llamada `f`, esto es, en la declaración:

```
int r = f(i);
```

Esta línea se aborta entonces y dado que reside en un bloque `try`, el programa saltará al primero de los manejadores de bloques `try`, que coincide con el tipo `error_desbordamiento` y que es el bloque `catch`, obteniéndose entonces:

```
No se puede dar resultado
```

Tan pronto como se alcanza un manejador que captura la excepción, ésta se elimina. La ejecución prosigue entonces normalmente con la sentencia que viene *después* del bloque `try`; es decir, la sentencia que viene después de la última de los manejadores.

Ejemplo 22.4

```
// archivo catch1.cpp
catch(char* mensaje)
{
   cerr << mensaje << endl;
   exit(1)
}
catch(...)
{
   cerr < " esto es todo pueblo " << endl;
   abort();
}
```

Está permitido que una signatura con puntos suspensivos se corresponda con cualquier tipo de argumentos. También, el argumento formal puede ser una declaración abstracta, significándole que puede tener información de tipo sin un nombre de variable.

El manejador se invoca en una expresión `throw` apropiada. En ese punto se sale del bloque `try`.

22.6. ESPECIFICACIÓN DE EXCEPCIONES

La técnica de manejo de excepciones se basa, como se ha visto, en la captura de excepciones que han ocurrido en las funciones que han sido llamadas. Varios métodos se pueden aplicar a tratar con diferentes tipos de excepciones. Una pregunta que cabe hacerse es: «¿Cómo se conoce cuál es el tipo de excepción que puede generar una función llamada?» Un método de conocer esto es, naturalmente, leer el código de programa de la función real, pero esto no es posible en la práctica cuando se aplica a funciones que son parte de grandes programas y las cuales llaman a otras funciones por sí mismas. Otro método es leer la documentación disponible sobre la función real y esperar que contenga información sobre los diferentes tipos de excepciones que se pueden generar. Desgraciadamente, tal información no siempre se puede encontrar.

C++ ofrece una tercera posibilidad. Una declaración de función puede contener una *especificación* de cuáles excepciones puede generar la función. Una *especificación de excepciones* proporciona una solución que se puede utilizar para listar las excepciones que una función puede lanzar con la declaración de función. Garantiza que la función no lance ningún otro tipo de excepciones. Los sistemas bien diseñados con manejo de excepciones necesitan definir qué funciones lanzan excepciones y cuáles funciones

no. Con especificaciones de excepciones se describen exactamente cuales excepciones, si existen, lanzan una función. También se tienen que disponer de controles para saber qué sucede si las funciones lanzan una excepción *«no prevista»*.

Una especificación de excepciones se añade a una declaración o a una definición de una función. Los formatos son:

```
tipo nombre_función(signatura) throw (e1, e2, eN); // prototipo
tipo nombre_función(signatura) throw (e1, e2, eN); // definición
{
   cuerpo de la función
}
```

`e1, e2,eN` lista separada por comas de nombres de excepciones (la lista especifica que la función puede lanzar directa o indirectamente, incluyendo excepciones derivadas públicamente de estos nombres).

Si una excepción diferente se lanza, el mecanismo de excepciones llama a `unexpected()`. Una función no lanza excepciones si la lista de excepciones es vacía. Se puede incluir una especificación de excepciones con cualquier función en C++, incluyendo funciones miembros de clases y funciones de plantilla.

Sintácticamente, una *especificación de excepciones* es parte de una declaración o definición de funciones.

Formato: `cabecera_función throw (lista de tipos).`

Si la lista está vacía el compilador puede suponer que no se ejecutará ningún `throw` por la función, bien directa o indirectamente.

```
void noexcep(int i) throw ();
```

Especificación de excepciones vacía

```
1. void g (parámetros) throw()  ←——  la función no puede generar
   {...}                              ninguna excepción
2. void g (parámetros)  ←——————————  la función puede generar
   {...}                              excepciones de cualquier tipo
```

Ejemplo

```
void f(parámetros) throw (T1, T2)
{...}
```

especificación de excepciones (señala a `throw (T1, T2)`)

Regla

Una especificación de excepciones sigue a la lista de parámetros de funciones. Se especifica con la palabra reservada `throw` seguida por una lista de tipos de excepciones encerradas entre paréntesis.

Normas de especificación de excepciones

- Una especificación de excepción se puede añadir al final de una declaración de función

```
tipo_retorno f (parámetros) throw(T1, T2, T3,...)
```

Entonces la función sólo genera excepciones de los tipos dados en la especificación.

- Una especificación de excepción vacía significa que ninguna excepción se generará por la función:

```
tipo_retorno f (parámetros) throw();
```

- Si no hay ninguna especificación de excepción, se pueden generar todos los tipos de excepciones:

```
tipo_retorno f (parámetros);
```

- Si se genera una excepción de un tipo no dado en la especificación de excepciones, se llama a la función `unexpected` y ésta acción termina el programa. La función `unexpected` puede ser reemplazada por la función no estándar:

```
set_unexpected (nombre_de_función_estándar);
```

Esta acción puede generar una excepción de tipo `bad_exception`.

Ejemplo

```
// Clase pila con especificaciones de excepciones
class PilaTest {
public:
   // ...
   void quitar (int &valor) throw (quitarEnPilaVacia);
   void meter  (int valor) throw (meterEnPilaLlena);
private:
   // ...
};
```

Ejemplo 22.5

En el prototipo de la función siguiente `f()`, se indica que `int` y `double` son los tipos de excepciones que se pueden requerir manejar si se invoca `f()`.

```
void f(char c) throw (int, double)
```

Si la función `f()` intenta generar una excepción diferente a los tipos `int` o `double` (o en general, tipos derivados de la lista `throw`), entonces esta excepción se corresponde con una invocación de la función `unexpected`, que por defecto termina el programa. Por ejemplo, supongamos que `f()` tenga la definición siguiente:

```
void f(char) throw (int, double) {
   if (isupper(')
      throw(1);
   else if (islower(')
      throw(1.0);
   else if (c == '.')
   throw(c);
}
```

Si la invocación

```
f('a')
```

se ejecuta, entonces se genera una excepción de tipo `double`. Si se invoca la función

```
f('A')
```

entonces se genera una excepción del tipo `int`. Si se ejecuta

```
f('.')
```

el programa se termina. Esto se debe a que se lanza una excepción de tipo `char`, donde el tipo `char` no es ni del tipo `int` ni `double`, ni se deriva de ninguno de los tipos `int` ni `double`.

22.7. EXCEPCIONES IMPREVISTAS

Las especificaciones de las excepciones representan las excepciones que una función puede lanzar bien directa o bien indirectamente. Si una función lanza una excepción que no aparece en la especificación de excepciones de la función, la excepción es imprevista (*unexpected*). Por omisión, el mecanismo de excepciones llama a `unexpected()`, que llama a `terminate()` para detener su programa.

La función del sistema `terminate()` se llama cuando ningún manejador se ha proporcionado para tratar con una excepción. La función `abort()` se llama por omisión. Inmediatamente termina el programa, devolviendo el control al sistema operativo. Se puede también especificar otra acción utilizando `set_terminate()` para proporcionar un manejador. Estas declaraciones se encuentran en el archivo *except.h*.

El manejador del sistema `unexpected()` se llama cuando una función lanza una excepción que no está en su lista de especificaciones de excepciones. Por omisión se llama a la función `terminate()`; en caso contrario se puede utilizar `set_unexpected()` para proporcionar un manejador.

NORMAS DE ESPECIFICACIONES DE EXCEPCIONES

- Una especificación de excepciones se puede añadir al final de la declaración de la función:

  ```
  tipo_retorno f (parámetros) throw(T1, T2, T3,...)
  ```

  En este caso la función sólo debe generar excepciones de los tipos dados en la especificación.

- Una especificación de excepciones vacía significa que ninguna excepción se generará por la función:

  ```
  tipo-_retorno f (parámetros) throw();
  ```

  Si no existe especificación de excepciones, todos los tipos de excepciones se pueden generar:

  ```
  tipo_retorno f (parámetros);
  ```

- Si una excepción de un tipo no dado en la especificación de excepciones, se llama a la función `unexpected` y esta acción termina el programa. La función `unexpected` se puede reemplazar por la función no estándar:

  ```
  set_unexpected (nombre_de_función_estándar)
  ```

  Esta acción puede generar una excepción del tipo `bad_exception.`

22.8. APLICACIONES PRÁCTICAS DE MANEJO DE EXCEPCIONES

Para ilustrar el manejo de excepciones en C++ examinaremos varios ejemplos:

22.8.1. Calcular las raíces de una ecuación de segundo grado

Aplicación 22.1

Tratamiento de las excepciones para la resolución de una ecuación de segundo grado.

Una ecuación de segundo grado de la forma $ax^2 + bx + c = 0$ tiene las siguientes raíces reales.

$$x_1 = \frac{-b + \sqrt{b^2 - 4\,ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4\,ac}}{2a}$$

Los casos en los cuales las expresiones anteriores no tiene sentido son: 1) $a = 0$; 2) $b^2 - 4\,ac < 0$, que no produce raíces reales, sino imaginarias. En consecuencia, se consideran excepciones de tipo error tales como:

```
enum error {no_raices_reales, coeficiente_a_cero};
```

Una codificación completa es:

```
#include <cstdlib>
#include <iostream>
#include <math.h>

using namespace std;
enum error {no_raices_reales, Coeficiente_a_cero};

void raices(float a, float b, float c, float &r1, float &r2)
throw(error)
{
      float discr;
      if(b*b < 4 * a * c)
            throw no_raices_reales;
      if(a==0)
            throw Coeficiente_a_cero;
      discr = sqrt(b * b – 4 * a * c);
      r1 = (–b – discr) / (2 * a);
      r2 = (–b + discr) / (2 * a);
}

int main(int argc, char *argv[])
{
    float a, b, c, r1, r2;
    cout << " introduzca coefiecientes de la ecuación de 2º grado :";
    cin >> a >> b >> c;
    try {
            raices (a, b, c, r1, r2);
            cout << " raices reales " << r1 << "  " << r2 << endl;
```

```
        }
    catch (error e)
          {
            switch(e) {
            case no_raices_reales :
                  cout << "Ninguna raíz real" << endl;
                  break;
            case Coeficiente_a_cero :
                  cout << "Primer coeficiente cero" << endl;
            }
      }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

22.8.2. Control de excepciones en una estructura tipo pila

La *pila* es una estructura lineal que recibe datos por un extremo y los retira por el mismo extremo, siguiendo la regla *«último elemento en entrar, primero en salir»*.

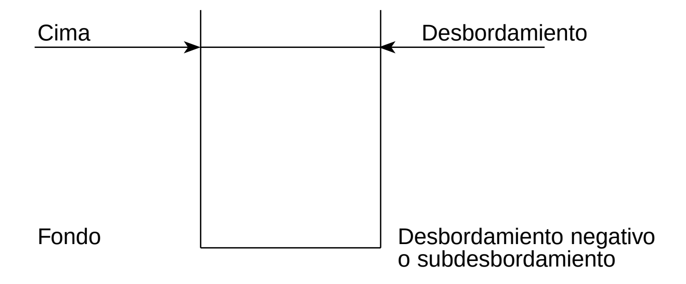


Figura 22.2. Estructura *pila*.

Declaremos una clase `Pila`, que es una pila de enteros con un máximo de diez elementos. La clase `Pila` declara dos clases anidadas, `Desbordamiento` (*owerflow*) y `Subdesbordamiento` (*underflow*), que se utilizarán para manejar las condiciones de error: *desbordamiento* (la pila está llena) y *subdesbordamiento* o *desbordamiento negativo* (la pila está vacía); en ambos casos, ni se pueden introducir datos en la pila (`Desbordamiento`) ni sacar datos de la misma (`Subdesbordamiento`). Cuando la pila se desborde en alguno de los dos extremos (cima o el fondo), se lanzan las excepciones apropiadas:

```
#include <iostream.h>
const TAMAPILA = 5;           // tamaño máximo de la pila

class Pila
{
public:
   class Desbordamiento       // una clase excepción
   {
   public:
      int valdesborde;
```

```
      Desbordamiento(int i) : valdesborde(i)  {}
   };

   class Subdesbordamiento   // una clase excepción
   {
   public:
      Subdesbordamiento() {}
   };

   Pila () {cima = -1;}
   void meter(int item)
   {
      if (cima < (TAMAPILA-1)) lapila[++cima] = item;
         else throw Desbordamiento(item);
   }

   int sacar()
   {
      if (cima > -1) return lapila[cima––];
         else throw Subdesbordamiento();
   }
private:
   int lapila[TAMAPILA];
   int cima;
};
```

El siguiente programa principal declara una pila y manejadores de excepciones para las condiciones de desbordamiento y subdesbordamiento de la pila. El programa fuerza a la pila para que se desborde, haciendo que se invoque al manejador de excepciones.

```
#include <iostream>
using namespace std;

void main()
{
   Pila mipila;
   int i = 5, j = 25;
   // Bloque try
   try
   {
      mipila.meter(i);
      mipila.meter(j);
      mipila.meter(1);
      mipila.meter(12345);
      mipila.meter(9999);
      // Pila llena con cinco números se fuerza una excepción
      mipila.meter(100);    // Lanza Pila::Desbordamiento
   }
   // Manejadores de excepciones
   catch(Pila::Desbordamiento &p)
   {
      cout << "La pila se ha desbordado tratando de meter :"
           << p.valdesborde << endl;
   }
```

```
   catch(Pila::Subdesbordamiento &p)
   {
      cout << "Se ha producido un rebose negativo" << endl;
   }
}
```

Cuando se ejecuta el programa, se visualiza:

```
La pila se ha desbordado tratando de meter : 100
```

RESUMEN

- Las excepciones son, normalmente, condiciones (situaciones) de error imprevistas. Normalmente estas condiciones terminan el programa del usuario con un mensaje de error proporcionado por el sistema. Ejemplos son: división por cero, índices fuera de límites en un array, etc.
- C++ posee un mecanismo para manejar excepciones, muy similar al del lenguaje Ada.
- El código C++ puede levantar (*raise*) una excepción utilizando la expresión `throw`. La excepción se maneja invocando un manejador de excepciones seleccionado de una lista de manejadores que se encuentran al final del bloque `try` del manejador.
- Sintácticamente, `throw` presenta los formatos:

```
throw
throw expresión
```

`throw expresión` lanza una excepción en un bloque `try`. `throw` sin argumentos se puede utilizar en una sentencia `catch` para relanzar la excepción actual.
- Sintácticamente, un bloque `try` tiene el formato

```
try
sentencia compuesta
lista de manejadores
```

El bloque `try` es el contexto para decidir qué manejadores se invocan en una excepción levantada. El orden en el que están definidos los manejadores, determina el orden en el que un manejador de una excepción levantada va a ser invocada.
- La sintaxis de un manejador es:

```
catch(argumento formal)
  sentencia compuesta
```

- La especificación de excepciones es parte de una declaración de función y tiene el formato:

```
cabecera función throw (tipo lista)
```

- El manejador que proporciona el sistema `terminate()` se llama cuando ningún otro manejador se ha previsto para tratar una excepción. El manejador del sistema `unexpected()` se llama cuando una función lanza una excepción que no está en su lista de especificaciones de excepciones. Por defecto, `terminate()` llama a la función `abort()`. El comportamiento por defecto de `unexpected()` es llamar a `terminate()`.

EJERCICIOS

22.1. El siguiente programa que maneja un algoritmo de ordenación básico no funciona bien. Sitúe declaraciones en el código del programa de modo que se compruebe si este código funciona correctamente. Escriba el programa correcto.

```
// Programa de ordenar
#include <iostream.h>

void intercambio (int x, int y)
{

      int aux = x;
      x = y;
      y = aux;
}

void ordenar (int 1[], int n)
{
      int i, j;
         for (i=0; i|=n; ++i)
            for (j=i; j|=n; ++j)
               if (1[j] < 1[j+1])
                  intercambio
                     (1[j],
                     1[j+1]);
      }

main()
{
      int z[12]={14,13,8,7,6,12,11,
         10,9,-5,1,5};

      ordenar (z, 12);
      for (int i=0-; i<12; ++i)
            cout << <[i] <<+ \t';
      cout <<-"\ ordenado" << endl;
      }
```

22.2. Escribir el código de una clase C++ que lance excepciones para cuantas condiciones estime convenientes. Utilizar una cláusula `catch` que utilice una sentencia `switch` para seleccionar un mensaje apropiado y terminar el cálculo. *Nota*: Utilice un tipo enumerado para listar las condiciones `enum error_pila {overflow, underflow,...};`

22.3. El código siguiente sirve para definir y manejar excepciones:

```
class cadena {
      char* s;
public:
      enum {minLong = 1, maxLong =
         1000};
      cadena();
      cadena(int);
      ...
};

cadena::cadena(int longitud)
{
// definir excepción fuera de lími-
   tes y lanzarlo
if (longitud < minLong || longitud
    > maxLong)
      throw (longitud);
s = new char [longitud];
// definir excepcion "fuera de me-
   moria" y lanzamiento
if (s == 0)
      throw ("Fuera de memoria");
}
...
void f(int n)
{
      try{
            cadena cad (n);
}

catch (char* Msgerr)
{
      cerr << Msgerr << endl;
      abort();
}
catch(int k)
{
         cerr << "Error de fuera de
            rango-:" << k << endl;
         if (cadena::-maxLong);
      }
      }
```

a) ¿Qué sucede si las líneas siguientes se eliminan de la función `f` y se invoca `f` con el argumento 5.000?

b) ¿Qué sucede si se cambia el constructor por el código siguiente y se invoca `f` con el argumento 5.000?

```
cadena::cadena (int n) throw
   (char*)
{
   ...
}
```

Recursos (libros y sitios web)

A1. LIBROS

[C++, 1998] X3 Secretariat: Standard – The C++ Language. X3J16-14882. Information TechnologyCouncil (NSITC). Washintong, DC, USA.

[Ellis, 1989] Margaret A. Ellis y Bjarne Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley. Reading, Mass. 1990. Versión española, *C++. Manual de referencias con anotaciones*. Reading Massachusetts: Addison-Wesley, 1994. (Esta versión ha sido traducida por los profesores Miguel Katrib de la Universidad de la Habana y Luis Joyanes de la Universidad Pontificia de Salamanca en Madrid.)

[Kernighan, 1988] Brian W. Kernighan y Dennis M. Ritchie: *The C Programming Language* (2.ª ed.). Prentice-Hall. Englewood Cliffs. New Jersey, 1998

[Lischner, 2003] Ray Lischner. *C++ in a nutshell*. O'Reilly, 2003.

[Stroustrup, 1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading Mass, 1986.

[Stroustrup, 1991] Bjarne Stroustrup: *The C++ Programming Language* (2.ª ed.). Addison-Wesley. Reading Mass, 1991.

[Stroustrup, 1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading Mass, 1994.

[Stroustrup, 1986] Bjarne Stroustrup: «A perspective on ISO C++», *The C++ Report*, vol. 17/8, octubre 1995 [`www.research.att.com/˜bs/papers.htm`].

[Stroustrup, 1997] Bjarne Stroustrup: *The C++ Programming Language* (3.ª ed.). Addison-Wesley. Reading Mass, 1997

[Stroustrup, 2000] Bjarne Stroustrup: *The C++ Programming Language* (Edition especial). Addison-Wesley. Reading Mass, 2000. Versión en español traducida por un equipo de profesores de la Facultad de Informática de la Universidad Pontificia de Salamanca campus de Madrid (España) dirigidos por el autor de esta obra.

A2. SITIOS WEB

1. Sitios web de C++

Existen cientos de miles de páginas Web referidas a C++ (el día de la última consulta del término C++: En Google 233.000.000 páginas; en Yahoo, 55.100.000) por lo que hemos seleccionado algunas de las más significativas atendiendo a la notoriedad e importancia del sitio en base al autor, organización, centro de recursos, etc. y que consideramos serán de gran utilidad para el lector en su fase de aprendizaje y sobre todo en su fase profesional.

- **Página web de Bjarne Stroustrup (creador de C++)**.
  `www.research.att.com/˜bs`

- **Cplusplus.com**
  `www.cplusplus.com`
  Página con gran cantidad de datos relativos a C++: tutoriales, información de compiladores, forum...
- **C++ FAQ Lite**
  `//parashift.com/c++-faq-lite/index.html`
  Página muy importante de preguntas realizadas más frecuentemente.
- **Cprogramming.com**
  `www.cprogramming.com`
  Tutoriales, herramientas, recursos, etc.
- **Acerca de C/C++/C#**
  `//cplus.about.com`
  Parte de un sitio web más completo sobre numerosos temas.
- **Guías de estilo de C y C++**
  `www.chris-lott.org/resources/cstyle`
  Reglas y normas para buenos estilos de programación.
- **Estándares abiertos**
  `www.open-std.org`
- **Comité de estándares de C++**
  `www.open-std.org/JTC1/SC22/WG21/`
- **C++ Standard: ANSI Draft/ISO Working Papers**
  `www.csi.csusb.edu/dick/c++std/`
- **ANSI/ISO C++ Professional Programmer's Handbook. Que**
  `www-f9.ijs.si/~matevz/docs/C++/ansi_cpp_progr_handbook/index.htm`

2. Otros sitios web de interés sobre C++ y lenguajes de programación

`www.msj.com/msjquery.html`
Revista Microsoft Systems Journal

`www.shareware.com`
Software shareware

`msdn.microsoft.com/developer`
Página oficial de Microsoft sobre Visual C++

`www.borland.com`
Página oficial del fabricante Borland

`www.lysator.liu.se/c/`
The Development of the C Language

`//en.wikibooks.org/wiki/Programming:C`
Programaming C en Wikibooks

Historia de C en la enciclopedia Wikipedia (10 páginas excelentes)
`//en.wikipedia.org/wiki/C_programming_language`

Página de Dennis M. Ritchie
www.cs.bell-labs.com/who/dmr/index.html

Preguntas y respuestas frecuentes sobre C (FAQ)
`www.faqs.org/faqs/C-faq/faq`
`www.faqs.org/faqs/C-faq/faq/index.html (de Steve Summit)`

Tutoriales

www.help.com/cat/2/259/hc/index-9.html
www.lysator.liu.se/c
www.anubis.dkung.dk/JTC1/SC22/WG14
www.uib.es/c-calculo/manuals/altrese/cursc.htm
www.help.com/cat/259/hc/index-9.html

Preguntas y respuestas frecuentes sobre C (FAQ)

www.eskimo.com|/~scs/C-faq/top.html
www.faqs.org/faqs/C-faq/faq
www.help.com/cat/2/259/hc/index-9.html
www.lysator.liu.se/c
www.anubis.dkung.dk/JTC1/SC22/WG14
www.uib.es/c-calculo/manuals/altrese/cursc.htm
www.help.com/cat/259/hc/index-9.html
http://www.parashift.com/c++-faq-lite/
http://www.cplusplus.com/

C99

www.comeaucomputing.com/techtalk/c99

COMPILADORES

An incomplete list of C++ Compilers (Bjarne Strustrup)

www.research.att.com/~bs/compilers.html

Thefreecountrycom
www.thefreecountry.com/compilers/cpp.shtml

Compilador GCC de GNU/Linux (Free Software Foundation)
//gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/

CompiladoresWin32 C/C++ de Willus.com
www.willus.com/ccomp.shtml

Compiladores e intérpretes C/C++
www.latindevelopers.com/res/C++/compilers

Compilador Lxx-Win32 C de Jacob Navia
www.cs.virginia.edu/~lcc-win32/

El Rincón del C
www.elrincondec.com/compile

Visual Studio Beta 2005
//msdn2.microsoft.com/library/default.aspx

3. Revistas de informática/computación de propósito general y/o con secciones especializadas de programación y en particular de C/C++

Específicas de C++

C/C++ Users Journal	www.cuj.com
Dr. Dobb's Journal	www.ddj.com
C++ Report	www.creport.com
Visual C++ Developer Journal	www.vcdj.com
Dr.Dobb's (en español)	www.mkm-pi.com

PC Magazine	`www.ppcmag.com`
Linux Magazine	`www.linux-mag.com`
PC World	`www.pcworld.com`
Java Report	`www.javareport.com`
SIGS	`www.sigs.com`
Java Pro	`www.java-pro.com`
PC Actual	`www.pc-actual.com`
PC World España	`www.idg.es/pcworld`
MSDN Magazine	`//msdn.microsoft.com/msdnmag`
Sys Admin	`www.samag.com`
Software Development Magazine	`www.sdmagazine.com`
UNIX Review	`www.review.com`
Windows Developper's Journal	`www.wdj.com`
Journal Object Orientd Programming	`www.joopmag.com`

4. Organizaciones informáticas especializadas en C/C++

ACCU (Association of C and C++ Users).
`www.accu.org/`

ANSI (American National Standards Institute).
`www.ansi.org`

Comité ISO/IEC JTC1/SC22/WG14-C.
`//anubis.dkuug.dk/JTC1/SC22/WG14/`
Comité encargado de la estandarización y seguimiento de C-99.

Comité ISO/IEC JTC1/SC22/WG21-C++.
`anubis.dkuug.dk/jtc1/sc22/wg21/`
Comité encargado de la estandarización y seguimiento deC++.

ISO (International Organization for Standardization).
`www.iso.ch/`
Organización de aprobación de estándares de ámbito internacional (*entre ellos de C/C++*).

Índice

Programación en



Algoritmos, estructuras de datos y objetos

Esta obra se ha escrito como libro de referencia y guía de estudio en un curso de ***Introducción a la programación***, con una segunda parte que puede utilizarse en cursos de ***Introducción a las estructuras de datos*** y a la ***Programación orientada a objetos***; en todos estos cursos se suele utilizar C++ como lenguaje de programación. Los objetivos fundamentales de la obra son:

- Énfasis fuerte en el análisis, construcción y diseño de programas.
- Resolución de problemas mediante técnicas de programación.
- Introducción a la informática y a las ciencias de la computación utilizando C++ como herramienta de programación.

Por ello, el libro se ha diseñado para enseñar a programar utilizando C++, aunque también pretende enseñar C++. Así, se tratará de enseñar las técnicas clásicas y avanzadas de programación estructurada, junto con técnicas orientadas a objetos. La obra pretende enseñar a programar utilizando tres conceptos fundamentales:

- **Algoritmos.** Conjunto de instrucciones programadas para resolver una tarea específica.
- **Datos.** Una colección de datos que se proporcionan a los algoritmos que se han de ejecutar para encontrar una solución: los datos se organizan en estructuras de datos.
- **Objetos.** Conjuntos de datos y algoritmos que los manipulan, encapsulados en un tipo de dato nuevo conocido como objeto.

Los apendices del libro, disponibles en el sitio Web asociado, tratan:

- **ANSI/ISO C++**
- **STL (Standard Template Library)**
- **C frente a C++**

El libro dispone de un sitio Web asociado, en la dirección **http://www.mhe.es/joyanes** en el que se pueden encontrar, además de contenidos adicionales al libro, materiales complementarios y ejercicios de práctica.

www.mcgraw-hill.es

The McGraw·Hill Companies